AN ANALYTICAL APPROACH TO COMPUTER SYSTEMS SCHEDULING

Robert Mahl

Utah University

Prepared for:

Rome Air Development Center
Defense Advanced Research Projects Agency

June 1970

DISCLAIMER NOTICE

THIS DOCUMENT IS THE BEST QUALITY AVAILABLE.

COPY FURNISHED CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

RADC-TR-73-107
Technical Report
June 1970

AN ANALYTICAL APPROACH TO COMPUTER SYSTEMS SCHEDULING

University of Utah

Sponsored by
Defense Advanced Research Projects Agency
ARPA Order No. 829

Approved for public release;
distribution unlimited.

The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U. S. Government.

Rome Air Development Center
Air Force Systems Command
Griffiss Air Force Base, New York

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1 ORIGINATING ACTIVITY (Corporate author)

University of Utah
Salt Lake City, Utah 84112

2a. REPORT SECURITY CLASSIFICATION

Unclassified

2b. GROUP

3 REPORT TITLE

AN ANALYTICAL APPROACH TO COMPUTER SYSTEMS SCHEDULING

4 DESCRIPTIVE NOTES (Type of report and inclusive dates)

5 AUTHOR(S) (First name, middle initial, last name)

Robert Mahl

6 REPORT DATE

June 1970

7a TOTAL NO OF PAGES

7b NO OF REFS

8a CONTRACT OR GRANT NO

AF30(602)4277

b PROJECT NO

c

d

9a ORIGINATOR'S REPORT NUMBER(S)

9b OTHER REPORT NO(S) (Any other numbers that may be assigned this report)

RADC-TR-73-107

10 DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11 SUPPLEMENTARY NOTES

Monitored by
Murray Kesselman
RADC (ISCE), GAFB, NY 13441
AC 315 330-2018

12 SPONSORING MILITARY ACTIVITY

Defense Advanced Research Projects Agency
Wash DC 20301

13 ABSTRACT

This report examines some aspects of the problem of allocating resources in a multi-programmed computer system. It first investigates to what extent the users might participate in resource allocation decisions; a system that dynamically determines the prices of services is advocated. A model is studied which yields a balanced set of programs in order to get a good simultaneous usage of the available system's resources. It also examines how resource utilization figures can affect the choice of equipment to be used at a computer installation and the choice of a swapping algorithm at system's design time.

DD FORM 1473

UNCLASSIFIED

Security Classification

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Graphics						
Computer Graphics						

AN ANALYTICAL APPROACH TO COMPUTER SYSTEMS SCHEDULING

Robert Mahl

Contractor: University of Utah
Contract Number: AF30(602)-4277
Effective Date of Contract: 20 May 1966
Contract Expiration Date: 30 November 1970
Amount of Contract: $10,535, 198.54
Program Code Number: 6D30

Principal Investigator: Thomas G. Stockham, Jr.
Phone: 801 581-8224

Project Engineer: Murray Kesselman
Phone: 315 330-2018

This research was supported by the
Defense Advanced Research Projects
Agency of the Department of Defense
and was monitored by Murray Kesselman
RADC (ISCE), GAFB, NY 13441 under
Contract AF30(602)-4277.

AN ANALYTICAL APPROACH TO COMPUTER SYSTEMS SCHEDULING

PUBLICATION REVIEW

This technical report has been reviewed and is approved

MURRAY KESSELMAN
RADC Project Engineer

ACKNOWLEDGMENTS

I am especially grateful to Dr. David C. Evans, of the University of Utah, who has patiently directed my research, restoring my confidence in many hours of frustration.

Dr. Jean-Yves Leclerc, of France, convinced me in 1967 that computer systems can be studied scientifically. As such, I consider him as the origin of my vocation.

Many thanks to friends, professors and students of the University of Utah, among them Denis D. Seror (with whom I had numerous fruitful discussions), and Duane B. Call, who helped to improve the wording and the comprehensibility of this document.

I also appreciate the relevant critics and comments which were made on earlier versions of this paper, especially those of Roy A. Keir and of Peter J. Denning.

TABLE OF CONTENTS

ABSTRACT*

This report examines some aspects of the problem of allocating resources in a multiprogrammed computer system. It first investigates to what extent the users might participate in resource allocation decisions; a system that dynamically determines the prices of services is advocated. A model is studied which yields a balanced set of programs in order to get a good simultaneous usage of the available system's resources. It also examines how resource utilization figures can affect the choice of equipment to be used at a computer installation and the choice of a swapping algorithm at system's design time.

*This report reproduces a thesis of the same title submitted to the Department of Electrical Engineering, Division of Computer Science, University of Utah, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

CHAPTER I

INTRODUCTION

I.1 The Role of Scheduling or Allocation in Multiprogrammed Computer Systems.

For several years, computer scientists have been faced with the task of organizing large information processing systems which many users may access simultaneously. In these systems, there are a number of physical resources (cells in core memory, peripherals, ...). At each moment, some of these resources are allocated to some users (it is implied that some of these resources are not allocated to anybody, some others to the system, and finally some of them to many simultaneous users--like a shared program segment).

What are the specific problems for these large systems?

1) PROTECTION: A user must be prevented from accessing a resource which is not allocated to him; or, equivalently a user should "see" only the resources that he is allowed to access. Several solutions which are more or less satisfying have been proposed in the last few years, detailed accounts of which can be found in [1].

2) DEADLY EMBRACE occurs when two or more processes are mutually blocking each other, in that each of them is demanding a resource that another posesses and does not want to release. The problem of avoiding deadly embrace has been solved satisfactorily, for instance by Haberman [2], who assumed that some facts could be known about a user (his maximum demand for resources) before any resources were allocated to him.

3) The scheduling or ALLOCATION problem itself (to whom the resources should be allocated if there is any conflict) is clearly separate from the problem of protection and can be completely separated from the previous problem, because deadly embrace is only lethal for certain kinds of demands which are not relevant to the allocator (essentially, access to shared tables and shared files).

Problem #3 is relevant to this study.

I.2 Evolution of the Problem of Allocating Computer Systems' Resources.

The first computers were run in a batch-processing mode. There were really two processes--the system and a program to be run by a user. If either of them asked for a resource which could not be allocated (for example, too much core memory), the user's program was simply aborted, and the next one loaded.

Later came the idea of time-slicing the utilization of the entire set of main resources (core memory, CPU and disk I/O). At the end of a time-slice, a user would be deallocated, and another user allowed to issue requests to the available resources. The entire system could be considered as just one big resource. Response time in such simple systems could be studied by queuing theory.

Another idea was not only to partition the time domain, but also the space of the resources. One user might have the right to use 10K of core and half of the CPU time while another might get 5K of core, half of the CPU time and the disk I/O. Space-slicing could be done either independently or concurrently with time-slicing. Again, if a user should ask for more than the resources which he was allowed to

access, his request would be denied.

Clearly, such partitioning leads to inefficiencies. Why should the system refuse to allocate an additional page of core if it is available, but falls outside of the originally allocated share of resources (partition) of the requesting user? This drawback led to the idea of having a more dynamic partitioning; the share of a user would not be determined by the system once and for all, but would depend on the user's current and future requirements for resources.

Dynamic partitioning of the resources, or as it will later be termed, dynamic space-sharing, itself leads to inefficiencies. For instance, suppose that a user should request the quasi-totality of the existing core memory, and that it would be available at the moment at which the request was issued. Then, if a few milliseconds later other jobs were to ask for core, they wouldn't get it; and as a result most of the system's resources would be idle until the large job would liberate some of its core. To avoid these inefficiencies, it is necessary to allocate the entire set of resources as a whole, rather than to allocate each resource independently. Note that queuing theory is already too weak to analyze this situation.

A conflict arises when two or more users simultaneously want to use the same resource. Scheduling can be defined as the art of solving such conflicts. The scheduler has to solve immediate conflicts (by giving to some user a priority over the others) and minimize the probability of future conflicts, without necessarily reducing this probability to zero (which would lead to rejection of many non-currently conflicting requests).

The space-sharing problem consists of finding a set of simultaneous users for the computer, which would efficiently use the system's facilities, and optimize a given criterion over a certain time interval. In other words, one looks for a balanced set of users. This is related to what shall be called (in chapter II) the indivisibilities in space of the user's programs. It is not possible, for instance, to allocate the CPU to a user without giving him some amount of core if he requests it. A further difficulty comes from the indivisibilities in the time domain. For instance, if a program receives some core memory and the CPU, it is not reasonable to instantly take back the core memory without leaving it some time to get swapped back onto the secondary memory.

Reservations and guaranteed service are other constraints the scheduler has to cope with. A user might want to get on the system only if he can be assured of having some portion of the drum for his files, and a certain portion of the CPU and core memory for his computations for a full one hour period.

Some other users might want to get more complicated contracts: for instance, to impose deadlines for some amount of service. The general scheduling problem for the system is to find a set of users, each user having requested some contract, such that each contract in the set can be fully satisfied regardless of any possible pattern of users requests. Such a set shall be called a compatible set of contracts. For a given system there exists some contracts or set of contracts for which it knows how to check user compatibility. This creates a structure of possible contracts. A goal of modern schedulers is to have a structure of contracts which would be as rich as possible, and possibly even extendible.

Note that the structure of recognizable compatible contracts may vary, depending, for instance, on the time of day, since at particular times some contracts might be more likely to be requested than others. If a contract has been accepted by the system, it might also reduce the size of the structures available for newly arriving users.

Pricing is related to the structure of contracts. If a user asks for a contract that most probably will not allow other users simultaneous allocation, he will be charged more than if he requests resources under a very common contract compatible with other users.

1.3 What is Wrong with the Current Approaches to Resource Allocation?

Schedulers of most current time-shared systems consider each user as belonging to one of a certain number of classes. The strategies of sharing the computer between these classes, or between users of the same class, are either rigid and do not leave much choice to the user. The system will try, for instance, to minimize the average response time for real time users, and to maximize the CPU utilization for batch users; but it will artificially determine to which class a user belongs, as a function of his past use of resources. The system designers will show, of course, that certain kinds of programs and certain patterns of service requests (examples for which the system has been designed), yield a reasonable trade-off between resource utilization, response time, and overhead.

Such methods, which compromise between generality of the situations which can be handled, and complexity of the allocations algorithms, are not criticized here, though it would often be possible to get more gen-

orality for less complexity than in the existing systems. Instead, a deliberately futuristic hypothesis is presumed wherein more complexity is allowable, and where it is reasonable to consider how the user wants to and could participate in allocation decisions.

By trying to completely automate the management of the computer's resources, and by attempting to hide the really available resources from the user, a considerable optimization potential, that is the knowledge of a user about his program's behaviour, is lost. Of course, most users who have very small programs, or who are just looking for errors in their programs, do not want to be personnally involved in resource management. However, for large programs which are run often, it is desirable that the programmer be able to influence the allocation algorithms, yielding improvements in computation time or in memory usage. The efforts expended in achieving these improvements could be more than compensated for by a savings in the cost of the execution of the program. Note that there is a small number of programs which are run very often which consume almost the entire machine (command interpreters, compilers, file handling routines, etc...).

Why not ask the user to give his advice for a certain number of trade-offs concerning the use of such things as memory, swapping resources, ...) An example of such trade-offs will be given later, in the case of a one-pass compiler (section II.2).

A review of the most current methods of analysis of resource allocation algorithms in time-shared systems will now be presented.

1.3.1 Job Shop Scheduling.

References: Industrial Scheduling [3]

Theory of Scheduling [4]

Most of the studies of scheduling have been done in operations research for the problem of n jobs which are to be run in m shops (the order of the shops can be either imposed or arbitrary). The number of jobs is thus finite, and the time spent by a given job in a given shop is known in advance. A job never cycles (never passes through the same shop more than once). The aim is to find a schedule (order in which each shop is going to handle the jobs), which minimizes, for instance, the average time spent by a job in the system.

The problem has also been investigated under assumptions which are a little bit more relevant to computer systems. The number of jobs is infinite. They arrive randomly spaced in time, and just stream through the shops; but the jobs are assumed to all have similar characteristics (same needs for resources). Some queue handling disciplines have been studied. The SPTF (shortest processing time first) strategy seems, invariably, to yield good results.

The current queuing theory methodology in time-sharing systems analysis, is an extension of the early job-shop analysis (see next section). Another extension was to study, by a combination of simulation and heuristic analysis, a job-shop kind of problem where jobs cycle through the shops. A typical study of this kind can be found in [4]. The flavor is to try to solve an exact model, where all data on time spent by an activity on a resource are perfectly known. It is hard to see how such techniques could be used for real-time computer scheduling.

1.3.2 Current Status of Queuing Theory.

The scheduling of time-shared computer systems has been studied in the light of queuing theory. Reviews of such methods can be found in [6,7]. This approach is not satisfying because it applies only to very simple problems. Most of the time the system is reduced to just one preemptible resource (a CPU or a disk), and a simple strategy like RR (Round Robin), or FB (Foreground Background) is studied; the response time of the system to a user who waits to seize the unique resource for a certain time period, is computed. These strategies are obviously of no help to our problem, because the bottle necks of a modern computer system are the sizes of the memories and the bandwidth of the channels rather than the speed of the CPU's.

Note that the author does not believe that statistics on the average user's behaviour should be used when creating a resource allocation algorithm. Such statistics can be very useful to predict the performance of a system faced with a certain set of users demands, and to determine an initial hardware configuration for a computer installation; however, to have a resource allocation algorithm based on some assumptions of the users characteristics might later prove to be quite dangerous.

1.3.3 Simulation Methods.

These methods have taken into account certain statistics about the users and a certain configuration of available resources, so that the results are valid only in the cases for which the simulations were run. The number of parameters whose effect should be checked is so

large that it is generally impossible to check the effect of each of them for several sets of values of the others. The real danger of simulation is to lose control of the cause and effect relationships under an enormous stack of results, and to misinterpret these results. Here again, it is believed that simulation should be of great help in choosing an actual hardware configuration which will efficiently run a chosen sample of programs, but not in selecting a scheduling algorithm which can be effective when the characteristics of the users change.

1.3.4 Heuristic Approaches.

Obviously, many scheduling algorithms proposed so far try to optimize the use of just one computer resource. Generally, it is the CPU (is [6], for instance), sometimes it is the core memory utilization ([7]), or the utilization of the channel between 2 levels of memory ([10,11]). This resource is considered to be "critical," and the algorithm watches its utilization very carefully. The other resources are considered as "auxiliary," and the only work made by the algorithm consists of seeing whether they are saturated or not.

More promising is Denning's approach [12,13], which will be discussed in section II.3.

The trouble with most time-sharing models is that they provide vague estimates of systems performances by merely using some overall statistics on the set of jobs expected to run while them [8,18]. Thus, they cannot be used for scheduling if the aim is to get balanced sets of users.

It is worthwhile to briefly mention a very isolated approach taken by a team at UCLA. Absolutely everything about the program's behaviour is supposed to be known and synthesized in a directed graph. Using this information, Bovet [15] studies memory and processor allocation for a program running in a multiprocessor system. This is an example of how a user could optimize his own resource allocation for a given environment, but the models of programs seem too sophisticated to allow the finding of a balanced set of users without excessive overhead.

1.4 Organization of the thesis

I believe that there are some reasons why a user should participate in the resource allocation decisions. This thesis investigates how this can be achieved. There are essentially two approaches:

1) The user can make the decisions himself, or,

2) he can give the system some information about his own behaviour, urgency, and possible trade-offs between the use of various resources, and let it make the decisions.

The system should only worry about conflicts, and try to minimize them. To prevent a user from monopolizing the whole machine, it is necessary to control him by invoking a pricing system. If a user wants more of a certain resource, he has to pay more.

The computer system is then considered as a market of resources. At each moment, the users bid for some single resource or some set of resources for a period of time. The system would allocate itself to a set of bidders in order to optimize its own profit (for instance, the

sum of the accepted bids). The prices of resources would be determined dynamically by the system, as a function of the load (density of demand).

The problem is later complicated because of indivisibilities, which do not allow allocation of resources independently from each other and independently from previous allocations. These questions are studied in chapter II. A scheme is also proposed for structuring the set of allowable contracts for the users.

Chapter III studies a model which has as a goal the prediction of the amount of future conflict over a certain time interval, wherein something is known about the behaviour of the users' programs. The predictive model is used to get a balanced set of users. Note that the choice of the information to be given to the scheduling algorithm concerning the users' programs is very important. If it is too complicated, the accuracy becomes questionable; if too detailed, the scheduler gets too complicated. If the information about future behaviour is too simple, it can't be of much help. I made a choice, and I investigated the corresponding mathematical model, with the central preoccupation of getting a system's algorithm simple enough so that it could work economically in real time on future computer systems where some computing power could be spent for schedules.

Chapter IV examines how, at system's design time, the prices of various kinds of memory and channel devices can help in the choice of a swapping algorithm. Two swapping algorithms are compared in the light of current trends in prices of facilities and users' demands. Chapter IV also illustrates how access time (latency) of a slow memory can be

traded for bandwidth (amount of information accessed per unit of time).

While turning the pages of this thesis, the reader may discover that he gets into more practical and particular situations as the chapter numbers increase. However, I expect chapter IV to be more readily accepted by actual systems' designers than chapter III, and chapter III, itself, more than chapter II. Note that one of my goals is to decrease the gap between theory and implementation in resource allocation algorithms.

I do not claim that I have solved "the" problem of scheduling computer systems. The last chapter of the thesis indicates some problems and models which would be worth investigating.

1.5 Trends in future computer systems, which are relevant to this investigation.

When designing resource allocation schemes to be used in future computers, an awareness of probable hardware developments is essential. It might be that the production costs of certain resources will be greatly reduced and allocation problems associated with these resources could possibly almost vanish due to the affordability of large quantities of them.

For instance, the average program on a current computer system costs much more for its memory usage than for its processor usage. This is going to be accentuated in the future, when programs will use more and more memory and also more interaction. As a consequence, it is less and less important to concentrate efforts in keeping the CPU's busy all the time, but we would like to spare fast memory.

The low price of the CPU's has another consequence. It is getting feasible to use more computing power in order to get an optimum allocation of the resources. In a big computer system, one processor (microprogrammed for more flexibility) could be dedicated to scheduling. Thus, scheduling strategies will no longer need to be as simple as multilevel queues. It is reasonable to assume that it will be possible, for example, to solve a linear programming problem in real time in order to get a better schedule (with some good heuristics).

The cost of one bit does not drop so fast, but the cost of an access is expected to decrease considerably. In other words, the bandwidth of the memory will increase. This is due to the advent of large scale integration. MOS or TTL memories will replace core, and the optimum size of an MOS module will be smaller than that of a core module. This consideration provides much of the motivation for chapter IV.

Is there going to be a trend towards multiprocessing or towards large computer systems? Large systems present several advantages: in particular to store large data banks to be shared by many users. Theoretically, large computer systems should also be preferred because their resources might be more fully utilized when there is a large number of simultaneous users, since the irregularities in the demand for resources are somewhat eliminated. But the final test will be whether the small user will get all the service facilities from the big machine that he found on the smaller computer. What will be the possibility of reserving some portion of the machine in advance? The user wants to know how fast his virtual machine works. He might also be in a hurry and expect to get better response by paying more for it. The

authors of schedulers with multilevel priority queues always considered the user as passive or inert: all users are equal, and a user cannot react to the service he is getting except by modifying his pattern of requests, or by simply leaving the system. Moreover, the pricing of computer usage was based on flat rates, which are, as shown later, not dynamic enough and thereby lead to losses of efficiency [16].

Another factor which tends to limit the size of computer systems is the existence of non-linearities in the overhead. If the total overhead grows faster than a linear function of the size of the system or the number of users, there is a critical size where it gets unbearable. For instance, if there are n processors accessing m memory banks, the complexity of the cross-bar switch is known to be proportional to n x m[17], and the time to solve conflicts is proportional to log n + log m, both terms introducing non-linearities. As far as allocation is concerned, our linearity criterion forbids us to spend more time or computing power to make an individual allocation decision on a larger system. This is a very drastic condition. Note that most smart page replacement algorithms (like Least Recently Used or Denning's working set) do not satisfy the criterion, while simple algorithms (FIFO, LIFO) do. Note that the swapping algorithm which will be presented in IV.2 does satisfy the linearity criterion, while the ones of IV.1 and of Chapter III do not.

CHAPTER II

PRICING AND RESOURCE ALLOCATION

II.1 Introduction.

Suppose that a coffee shop was serving ice-cream to people on a first-come-first-served basis, without asking them to pay for it. As the news passed through the town, an enormous queue of children waiting to get their ice-cream was formed outside of the shop. Some of the children, after getting a first ice-cream, were going back to the end of the queue and waiting for a second one, and so on. When the shop started asking a quarter of a dollar in exchange for an ice-cream, the queue vanished.

It seems that computer scientists were slow to find out that a computer system is just a service. If it is given for free, there is a tendency towards misuse and efficiency is lost. I expect this to get more obvious as the extraordinary growth of the computer industry slows down.

Allocating resources was defined as solving conflicts between simultaneous requests for the same facilities. But wouldn't it be better to just avoid those conflicts by pricing the resources high enough so that the number of them is greatly reduced?

There are two possible philosophies in relating resource allocation to an economic system; they are given here for the case in which there is only one resource, but they can be generalized to more complicated situations:

1) The "a priori" pricing philosophy. The system chooses a price for the resour . As soon as a customer arrives, he gets the resource, provided that he wants to pay the price and that the resource is still available. From time to time, prices are adjusted, depending on variations of the offer and demand levels. This was advocated, for instance, by Nielsen [16].

2) The bidding philosophy. Each user submits a bid for a resource. At a certain time (chosen by the system), the resource is given to the highest bidder. In Sutherland's yen system [18], the previous bids are known by all the users. In the case of a real-time bidding, the bids would be secret (essentially to avoid the overhead of letting the user consult the currently expressed bids). This, however, does not mean that the user would pay the full amount of money that he offered; in fact, I suggest in II.2.4 that the user should only be charged the minimum amount that he would have had to offer to get the resource.

The bidding philosophy has two advantages over a priori pricing and one drawback:

1) The highest bidder always gets the resource (and not the first to arrive).

2) The bidding itself automatically determines the price to be charged to the user, so that no price adjustment is necessary.

3) However, with the bidding method, the user doesn't know whether he is to get the resource until the time arrives for the auction to be closed.

Note that both philosophies can be combined in the following way. The system sets a price at a certain level above the average price at

which the resource is expected to be sold. A user has the choice of reserving the resource immediately when he asks for it, at the price set for it, or taking the chance of waiting until the time has arrived where, if the resource has not yet been allocated, the bids are examined and the resource allocated and priced in the second way.

II.2 User made decisions in a system with decentralized control.

II.2.1 Some definitions.

Facilities. There are several kinds of resources (or facilities) in a computer system. For instance, a certain number of bits in a storage device is one resource, while accessing (writing, reading, or executing) these bits is another resource. A piece of software (like a compiler) can itself be considered as a resource, which can be bought or rented for money, but in the following developments only the hardware resources will be considered.

Note that anything demanded by the user can be called a resource (execution of a programmed operator, having a certain program in core memory, etc...). Ultimately, money is itself a resource; this notion will be useful in a later section.

User and System. The user is not just the human being who programs the computer; it is an independent decision-making entity, composed of the human and his programs, and even so-called "system's routines" that another part of the user has decided to activate.

The system is a particular user which makes resource-allocation decisions. Computer operators, managers, and basic system's programs

belong to the system.

II.2.2 Decisions made which are relevant to the user.

The sophisticated user might want to make some decisions which are ordinarily made by the system. The first idea which comes to mind is that the user desires to optimize himself relative to his environment.

Suppose first that each resource has a price, is available in any amount, and that the user tries to minimize the unit cost of running in such a system. Later, it will be seen that, in general, such a reasoning is too simplistic, because the users demand is not small relative to the total amount of resources in the system. Economists say that the demand is not atomic.

What are some of the decisions which could be made by the user?

1) Size of his working set of pages in first-level memory (in a paged system).

2) Choice of an urgency (total bid in a bidding system, or priority level in a system where a cost is associated with each priority level).

3) Size desired for I/O buffers.

4) Residency of a given segment at a certain memory level. Residency of a file at a certain memory level.

5) Collecting and eventual giving to the system of some statistics on a program which is often run. The idea that a user's knowledge is important in order to get better paging has been advocated, for instance, in [19,20]. Denning's demonstration [12] that his "working set" al-

gorithm performs better than L.R.U. (Least Recently Used), can be reinterpreted by saying that the inferiority of L.R.U. is that it handles all users in the same way.

The user, as we have defined it, could always call some shared system's routine to do his job. However, this routine would work under responsibility of the user, and he would be charged for the resources consumed by this routine.

The next subsection will show an example of how trade-offs between the usage of several resources can be known and serve the user to improve the resource usage of his program. In other cases, however, such trade-offs cannot be trusted to the users, because the same strategy has to be used for all programs in the system. Such a situation is analyzed in section IV.3 (chapter IV).

II.2.3 An example showing trade-offs between the use of several resources by a user: a one-pass compiler.

A fictitious compiler has 5 parts (or segments), each of which has certain characteristics as far as the locality of the program references is concerned.

(1) Syntax analyser and lexicographical analyser

20% of the memory references, and occupies 0.1K words

(2) Error messages

2.1% of references, 4K words

(3) Identifier table

15% of references, 8K words

(4) Code and data segments currently generated

4% of references, 4K words

(5) Syntax tables and semantic routines

60% of all references, 8K words

These 5 types of segments have very different properties:

(1) Each time (1) is accessed, almost all of its words are accessed.

(2) is accessed very seldom.

(3) has about 8 to 10 words of information per identifier, each of which is accessed about once or twice each time the identifier is accessed.

(4) is accessed quite randomly: two consecutive accesses are never contiguous or very close in time, and not always contiguous in space.

(5) Accesses to (5) are frequent, but not very correlated in time.

The memory system has 1) fast registers accessed in .1 microsecond, and 2) core memory accessed in 1 microsecond. The hardware allows, for instance, swapping pages of 32 words between the two levels of memory, with a replacement algorithm of the "working set" type of Peter Denning [13]. The size of the available memory is supposed to be larger than what the user might request.

Assume that there exists a pricing system for the computer resources, with a price p_1=10 for a memory cell of type 1, and p_2=1 for a type 2 memory cell, per unit of time. In this simplified model, the costs of the CPU and of the swapping bus between the two memories are supposed to be negligible. This system of prices is supposed to be quite stable, and the user can assume that it will not vary more than

slightly over a rather long period of time. The user has to determine optimum residency or swapping strategies for pages of his various segments. This assumes, of course, that the hardware is able to recognize, for each user's segment, which strategy should apply to it.

The detailed computations underlying this example are not shown here; however, they indicate, with some restrictive hypothesis, that the optimum strategy has to leave (1) resident in fast registers, (2) and (4) resident in core memory, and (3) and (5) should have pages swapping between the two levels, with a working set size of about 32 references. If fast memory had been much more expensive (overloaded system), (2), (3) and (5) should have been resident in core memory. The point is that it is possible to linearize the average cost of one reference for a given segment and a given strategy, in the form:

$$C = C_0 + a_1 p_1 + a_2 p_2$$

where p_1 and p_2 are the prices of the resources, and a_1 and a_2 are coefficients which depend on the chosen strategy for the segment. This allows quite fast determination of the best strategies for given costs of resources, before running the compiler. But let it be stated again that this optimization job is relevant to the user and not to the system (the concept of user, of course, includes programs working for the user).

Note that if, for the best possible set of strategies, the average cost of a program reference is too high, the user might decide to delay his run (the threshold might be a function of the urgency of the job). In Figure II-1, a strategy is represented by a point whose coordinates are the resource utilizations of the strategy. Strategy 3 is optimal

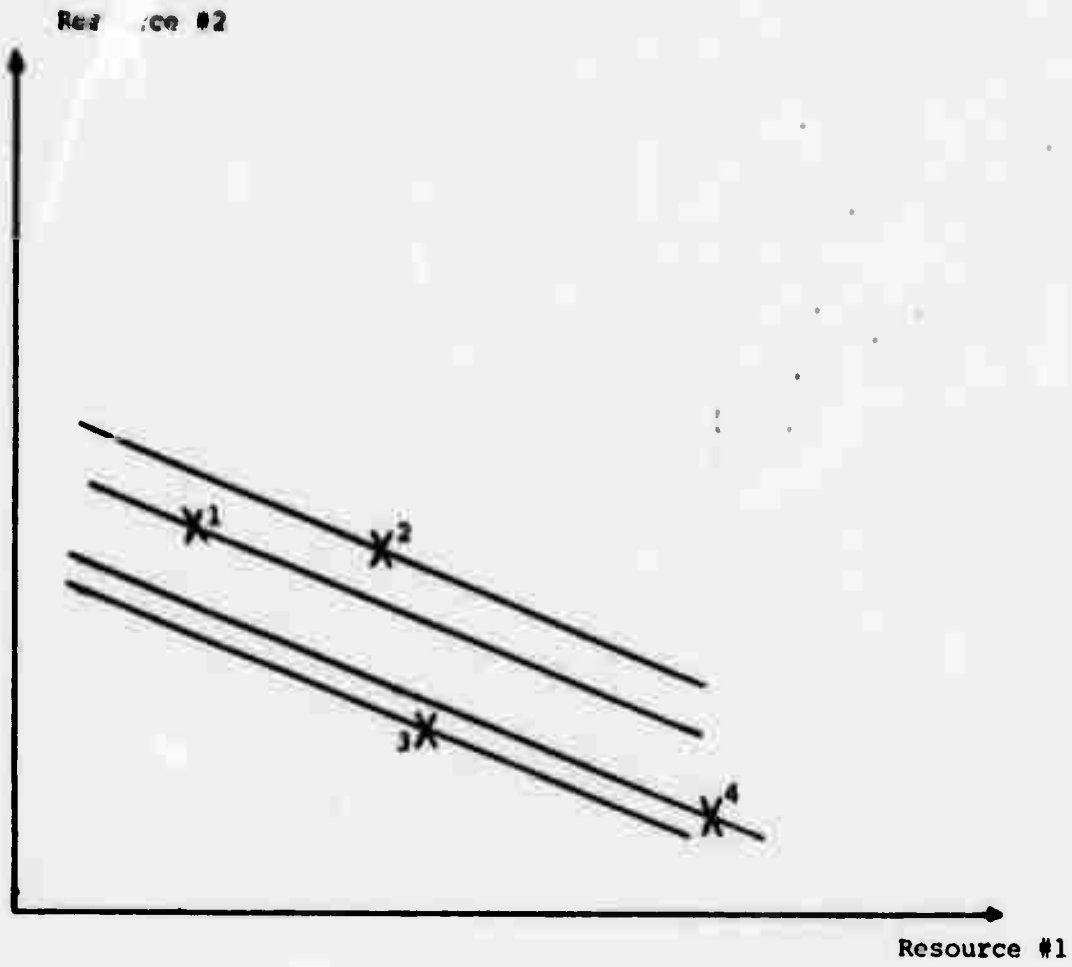


Fig. II-1

Trade-offs between several possible strategies
Strategy #3 is optimal (lowest cost)
The parallel lines join points of equal cost.

because its cost, which is a linear function of the resource usage, is minimum. Note that this figure does not pertain to the compiler, but is just an illustration of some possible strategies.

II.2.4 Pricing.

Modern economists like Debreu [21] have shown that a wrong pricing of resources leads to inefficiencies and loss of potential power. They advocate a system of marginal prices. For instance, if the demand for computer resources is low during night, the price should be correspondingly low. If there is just one user on the system (no conflict in demand), the price should be just equal to the marginal cost of keeping the system running (the cost of the operators, plus of electricity).

I believe that the following points characterize a fair system:

1. It will always sell a resource at a marginal cost (see point #4).

2. It does not make any distinction on behalf of the user (name of the user or previous history). In particular, if the user is willing to pay, there is no reason to penalize him even if he has used many of the facilities of the system in the recent past. In other words, there is no implicit priority system.

3. The system will never charge more than the user announced he wanted to pay. Nevertheless, the user might be given the resource at a price smaller than this maximum.

4. The general rule for allocating resources and charging for them is the following: the system takes the allocation decision which

maximizes its profit, but nevertheless it charges the individual user the minimum amount of money that this user would have had to offer to get the resource; all other bids being unchanged.

5. An immediate consequence of rule 4 is that two identical resources will cost the same at the same time, independent of the users they are allocated to.

6. If a sharable resource is available during a time-slice, then any user may use it without any cost to him. This takes care of the reentrant routines, for which only one user pays at a time (the first user to get the routine in core).[1]

7. The system must distinguish between conflicting demands for a resource (most of the practical cases), and cooperating demands (for instance, a reentrant piece of code). In the latter case, the different demands are considered as only one, with the maximum amount of money being the sum of what each user wants independently to spend for this resource.

A general idea underlying this thesis is that marginal pricing will have a good effect:

1) By trying to get better response at a lower price, the users will increase the system's efficiency rather than work against it (the problem of counter-measures has been reviewed in [22]).

2) Statistics will be provided to aid the users in estimating their chances of getting the desired response for a given amount of money at various hours of the day.

---

[1] A more complex model could be imagined under which the cost of the resource would be shared by the participating users.

3) Statistics will show the systems managers in which equipment lies a bottle-neck or which equipment is not really needed.

There is another reason for marginal pricing in a system where the same resource is reallocated very often. Without it a user at an open auction would automatically arrive at marginal price anyway, by slightly increasing his bid until he got the resource (or the bid reached the limit of what he wanted to pay). In any case, the overhead implied by such a strategy may be avoided by assuring the user of a "fair" price even if he immediately submits his maximum bid.

II.3 Indivisibility in space of the user requirements.

The previous paragraph handled cases where the demand could be considered as being

1) atomic

2) for a resource which could be allocated independently of any others, and independently of any previous or future allocation of the same resource.

Alas! This is not true, in general. It is impossible to allocate just 1K of core to a program asking for 3K; better not to allocate any resource at all to that program.

In this section, is considered the indivisibility in space, where the space considered is the space of the resources. A given user asks for a set of resources, for instance, for the duration of a time-slice. Now, consider how the system reacts under both the pricing and the bidding philosophies.

1) Under the pricing philosophy, a user is allocated as soon as

he asks for his set of resources, if there are enough available resources to satisfy him.

2) Under the bidding strategy, all users are allocated at the same moment, and the system tries to allocate a set of users in a way such as to maximize some economic criterion. If one tries to achieve a balance policy (i.e., to allocate an equilibrated set of users, to better use all the resources), then the bidding philosophy has to be adopted.

Denning [12] has proposed to formulate the allocation problem as a 0/1 integer linear programming problem (also known as a multidimensional Knapsack problem). If user #i asks for an amount a_{ij} of resource #j, then the system has to find a set S of users such that:

$$\sum_{i \in S} a_{ij} \leq A_j \qquad (\forall j)$$

where A_j is the available amount of resource j. The economic criterion (cost function) has the form:

$$E = \sum_{i \in S} c_i \quad , \text{ where } c_i \text{ is the bid of user \#i.}$$

The resources are, for instance, core memory and CPU. Suppose that one user asks for 25% of the CPU and 50% of core during a certain time interval, and another user for 70% of the CPU and 40% of the core. Clearly, if both of them are allocated, they will not take more than 90% of core and 90% of CPU (see figure 2), and thus they can be allocated.

This solution to the space indivisibility problem is not entirely satisfying, because a compute-bound user might request 100% of the CPU, and so should be alone in the system. However, suppose that this user

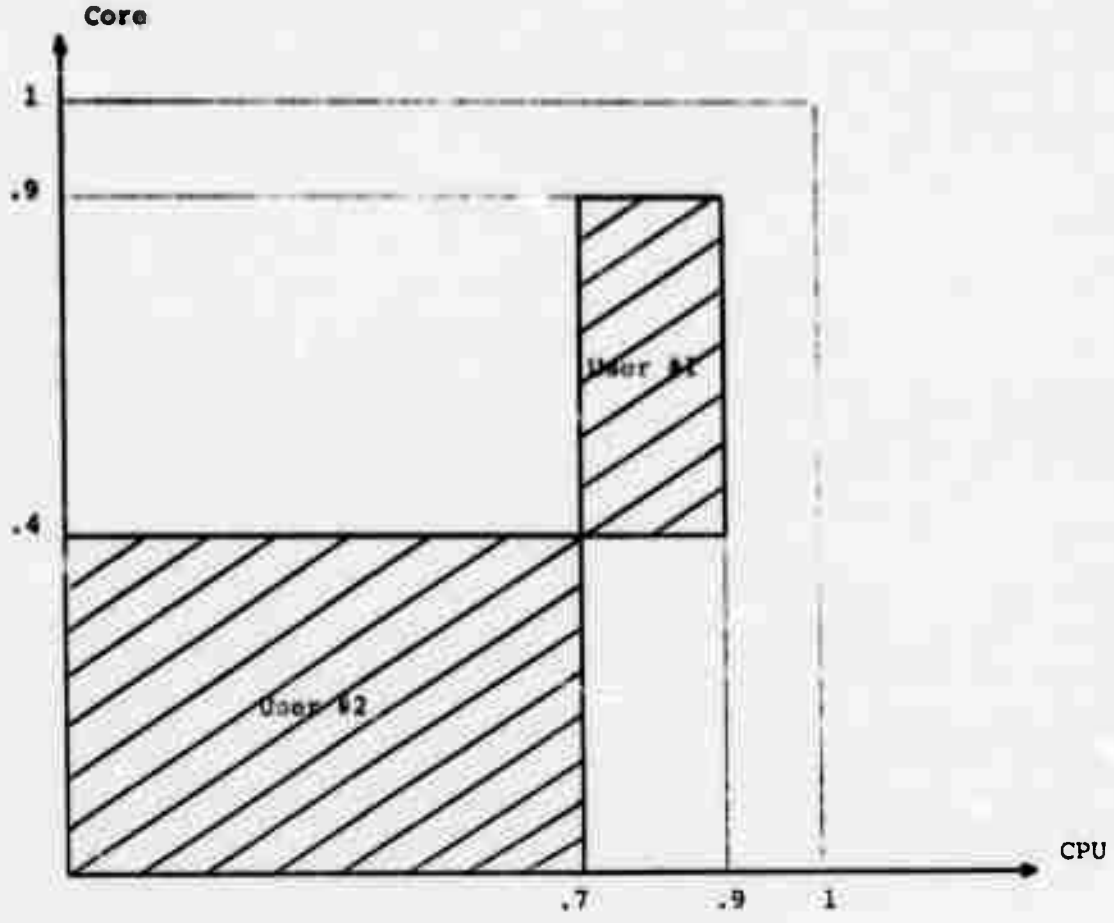


Fig. II-2

Multidimensional Knapsack Allocation

had only requested 50% of the CPU time. Another user could then have been allocated (for instance an I/O bound user who requests the CPU quite seldom). This new solution would be much more optimal. In other words, to use the terminology introduced in chapter II, the progress rates should not be determined by the users, but by the system.

Note that, even if the constraints are satisfied, the allocated users are not guaranteed the service that they requested. For instance, suppose there are 2 users, each of which asks for 45% of the CPU and 45% of the disk channel. Together, they ask for only 90% of both resources, but there can not be an a priori expectation that one job will use the CPU while the other is using the channel. If there is really bad synchronization between the two jobs, they will often both ask for the CPU at the same time or both for the channel, so that there will be little overlap between them. Chapter III of this thesis studies a model of such situations, and extends Denning's multidimensional Knapsack formulation to take care of them. Also in chapter III is given an algorithm to get an approximate solution of the multidimensional Knapsack under the special circumstances involved. This method of solving the Knapsack has the interesting peculiarity of leading to fair prices for the resources and the sets of resources allocated to the users. In this way, the system can keep statistics of these prices and use them as stated above. For thorough treatment of the Knapsack problem, see [23] and [24].

The "a priori" pricing of resources is also possible in a climate of space indivisibility. The idea is the following: If there is a cer-

tain set S of resources available, the system might expect to sell it at the average price P(S). If a user asks for a set S' of resources, the system would sell it to him at a price

$$c = k\left[P(S) - P(S - S')\right]$$

where S - S' is the set of resources remaining after resources of set S' have been allocated, and K is a constant greater than or equal to 1.

The precise function, P(S), has to be determined experimentally, by adjusting it to observed profits. Suppose, for instance, that there are two resources; the CPU and the core. P(x,y) is the average profit the system will make out of a percentage x of the CPU and y of core. Clearly, P(x,0) = P(0,y) = 0, because it is not possible to sell CPU without core or conversely core without CPU. An example of such a function is the cone represented in figure II-3. Its equation makes it homogeneous (first degree) in x and y:

$$P(x,y) = \sqrt{x\,y} \times \frac{ax + by}{x + y}$$

The coefficients a and b have to be adjusted by the system from its own experience.

II.4 Indivisibilities in time domain and reservations.

As an example of time domain indivisibilities, suppose the following. A program is in core and uses the CPU. The CPU resource can be instantly taken from this user, but not the core resource lest the user's job be destroyed! The user must be left in core at least for the period of time required to swap him out onto secondary storage.

The problem of reservations is somewhat similar. The user who comes to a console wants to be sure that he will own the console for

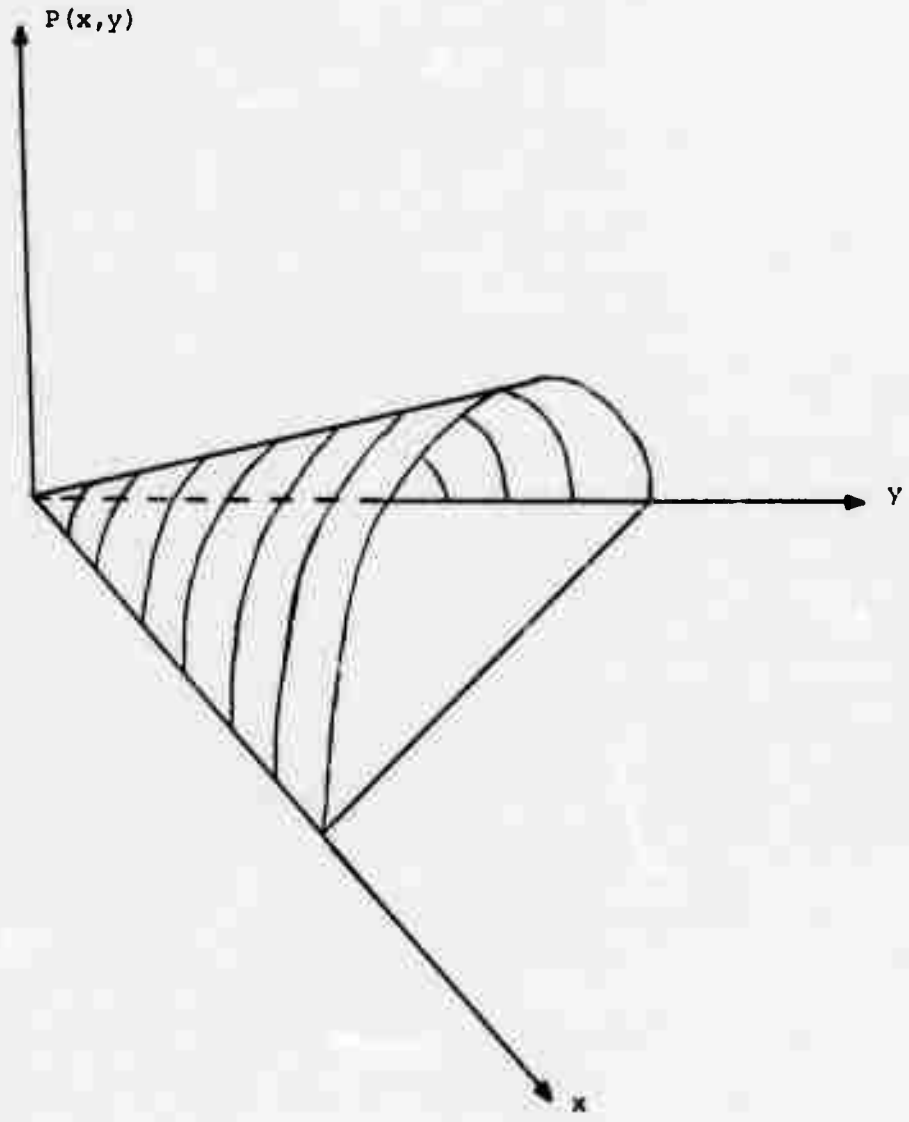


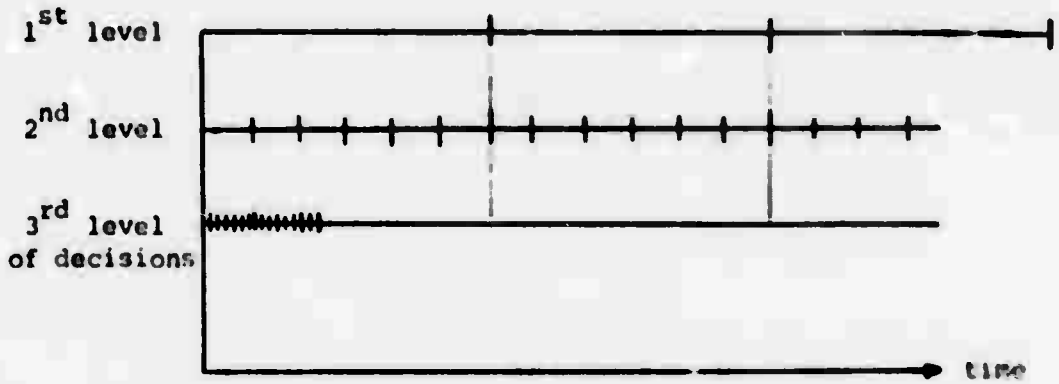
Fig. II-3

$$P(x,y) = (ax + by)\ \frac{\sqrt{xy}}{x+y}$$

at least a one hour period. He also wants to be sure that he will get 75K words on the disk for his files, and at least 3% of the usage of a compound resource (CPU • 10K core).

It will now be shown how this example can be handled by creating a structure of successive levels of allocation. The following ideas have to be applied:

1) Partial allocation decisions should be made for several time intervals, some of which are within some others, building a hierarchy in time. For instance, in our example, the decision of allocating the console and some part of the disk is made for a one hour interval. However, the decision of allocating the CPU and some core to a running program is made only for a one second interval.[2]



2) Resources have to be pooled in order to allow the user to buy a percentage of the pool in advance (1st level allocation), without knowing at that time exactly when he will use his buying power (2nd level decision).

---

[2] I am grateful to Professor Herbert Simon, of Carnegie-Mellon University, for having convinced me, in a private discussion, of the importance of multi-level scheduling.

The user buys, at the first level, a potential power to be used at the second level. This potential power can itself be called a money (or a resource), which is only valid for bidding or buying at the second level and during the time duration for which the first level decision was made. Because this resource exists in a limited amount only, the user is guaranteed, at the moment he buys it, that he has a certain percentage of it, and thus a guaranteed level of service.

Note that the money which was used to buy at the first level cannot be used to buy at the second level. One has first to buy the intermediate type of money.

The structure of allocation created above is a hierarchy of resources, which takes the form of a tree (figure II-4). Each resource can be used exclusively to buy resources which are under it in the tree, and only during the time interval for which the allocation was made at the level above. With each node of the tree is associated a set of rules, which tell how it can buy or bid for the resources which are under it in the tree. Note that some part of a resource might be at some node of the tree, while some other part might be at some other node. For instance, a part of core memory might be available to run time-shared users under a certain kind of contract, while another part of core might belong to a separate real-time user.

II.5 More involved contracts.

Real-time users might want to get a certain amount of service before a particular deadline. This can be handled either with a bidding system, where the user gradually increases his bids for systems resources

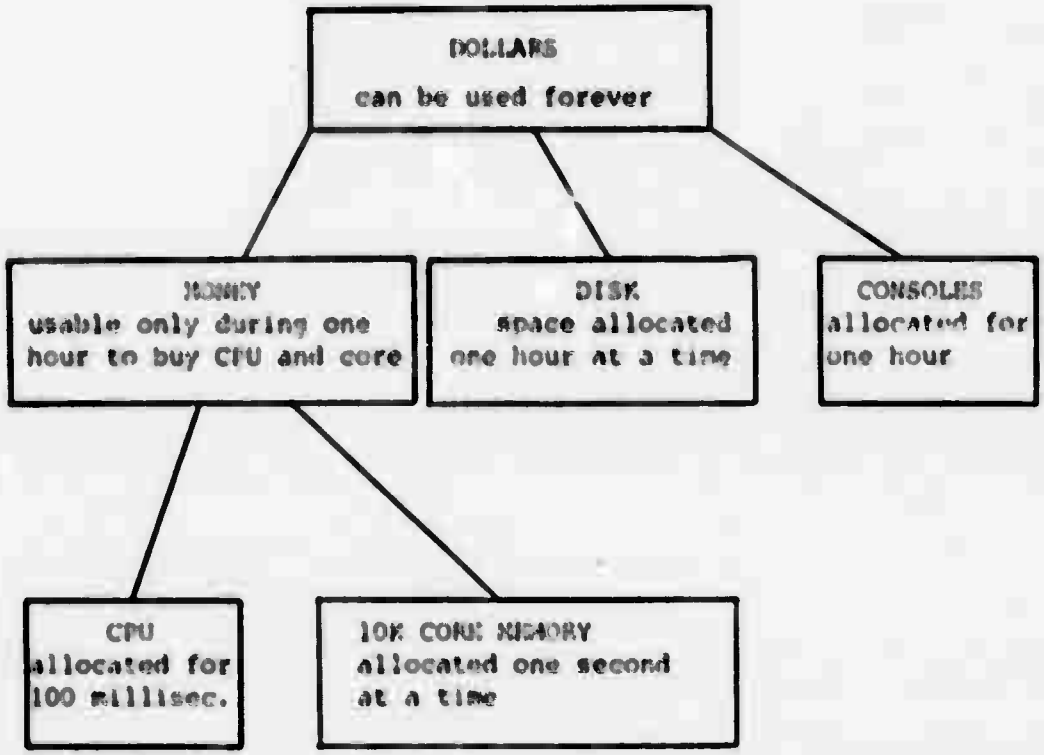

DOLLARS
can be used forever
MONEY
usable only during one hour to buy CPU and core
DISK
space allocated one hour at a time
CONSOLES
allocated for one hour
CPU
allocated for 100 millisec.
10K CORE MEMORY
allocated one second at a time

Fig. II-4

when the deadline approaches [25], or with a special contract, where the system takes the responsibility of finding out whether a given user can be satisfied within the current structure. Note that the scheduling method suggested in chapter III can be extended to take care of such users.

CHAPTER III

AN ANALYTICAL MODEL OF SPACE SHARING

III.1 Introduction.

This chapter attempts to show that better schedulers could be designed if some model of the interference[3] between users' requests for facilities were available. A program is a string of references[4] to certain resources. Whenever a program references a resource which is already totally allocated, a conflict occurs and the scheduler has to decide which program will get the resource. It will be shown how this can be done in a "pseudo-optimal" way by taking into account some information which can be made available before run-time on the patterns of the users' requests for facilities.[5]

Sections III.1 and III.2 introduce some of the concepts and terminology used later in the chapter. Section III.3 studies a model of "worst possible" synchronization between the users' requests for facilities under certain assumptions; among which is the assumption that resources are preemptible and each user has a fixed priority for accessing a resource (a given user has a different priority for each resource). In section III.4 the way in which the previous model can be used for scheduling is examined by formulating the resource allocation problem

---

[3] level of conflicts over some period of time.

[4] demand string

[5] this information, as we shall see, is related to the predicted usage ratios of the various resources by each individual program in the system.

as a Mathematical Programming Problem (finding the maximum of an economical function with certain constraints on the variables). Then the design of an algorithm which yields a nearly optimal solution to the M. P. problem is given. Section III.5 compares the model of section III.3 with models of some other scheduling strategies. The possible fruitfulness and extensions of this study are discussed in III.6 and III.7.

There are some resources, like the CPU, which can be preempted (transferred from one user to another) with less overhead than some other resources (like core memory, if the previous user has to be swapped). This leads to the idea of having a hierarchy in time of partial scheduling decisions; some decisions being made for smaller time intervals than others (see sections II.4 and III.4.1). The model which is studied in section III.3 will be used in section III.4 to relate two levels of scheduling. Whenever a decision is made for a long time interval (macroscheduling), the scheduler takes into account some information on the future demand pattern of the user during this time interval. The macroscheduler then sets some parameters of the lower level scheduler (microscheduler).[6]

Of course, solutions to macroscheduling problems depend on some information being available on the patterns of the users' requests for facilities. Haberman [2] has shown that such information can be useful in avoiding deadly embrace of processes in a time-shared environment. This information might be provided either by the user himself, or

---

[6] Note that microscheduling can be done by hardware, which, for instance, resolves conflicting requests for a memory bank. Then the software "sets" the hardware.

extrapolated from statistics collected by the system.

What information would be useful? It should be relevant to the scheduler by permitting computation, for instance, of the maximum possible "interference" between different jobs. It should be simple and condensed, because the scheduler has to operate rapidly, and finally this information should be easily available and characteristic enough of a given program that it could be used without any modification for several runs of the same program with different input data. In this paper, the information used will be the proportion of usage of the various preemptible resources[7] over a rather long time interval during which a job is to run, and the total (maximum) amount of non-preemptible resources required by the job.

Do we really need macroscheduling separated from microscheduling? In a recent paper [29], Stevens examines what was wrong with the Chippewa Operating System; he concludes that there were two flaws. First, the absence of a macroscheduler: the Chippewa system allocated resources for an indefinite period of time, without taking into account the global demand of each job. Thus, there was no guarantee when a job was allocated, that the job would not ask later for more memory than was available, and in this case the Chippewa scheduler did not take back the resources (CPU,...) already allocated. The second problem of Chippewa was that I/O bound jobs, or compute bound jobs, were not recognized as such by the scheduler, and so this information was not taken into account in assigning priorities for the resources. We will see

---

[7] Preemptible resources are those allocated by the microscheduler. For the moment, the reader might imagine the CPU as opposed to memory.

that better simultaneity in resource usage and between jobs' progresses is achieved by assigning a high priority for a resource to a job which will make little use of this resource.

III.2 Some Definitions.

User: An entity which requests and seizes resources, and which might also give some information about its future resource requirements. In this study, the words user, job, program and process describe the same concept.

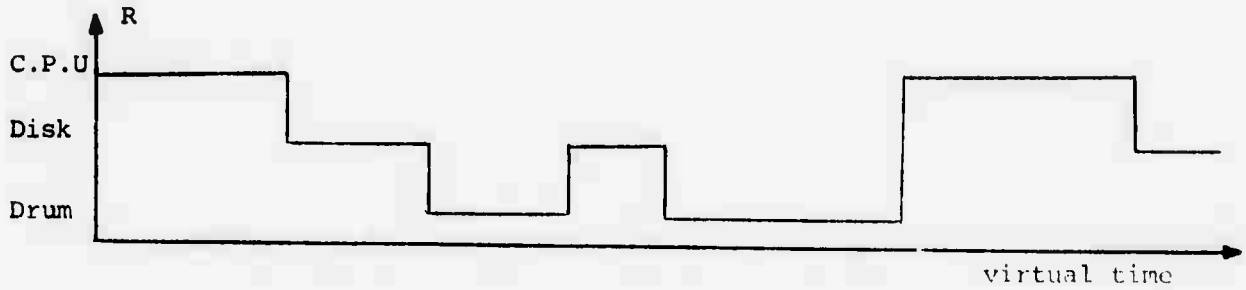
Demand string of a user: A program will be considered as a sequence of calls to various preemptible resources: CPU, I/O, ... such that one and only one resource is called at a time by a given program (no double buffering, for instance). This limitation could be removed, but helps to simplify the presentation.

Let R be the set of resources. A program is then some string $r_1 r_2 \ldots r_k$ over R, where r_i means that the user called on resource r_i as the i^{th} resource call. This notion is similar to what Denning used in more restrictive frameworks to describe page reference strings.

Virtual time of a user: During a certain real time interval ΔT_R, a user will get the resources requested in his demand string during a total interval of time ΔT_V. We define ΔT_V as the virtual time interval corresponding to the real time interval ΔT_R. Virtual time of a user normally runs slower than real time, but if a user were to permanently have top priority for accessing all resources, then virtual time for that user would be equivalent to real time.

The virtual time diagram of a user is a diagram in which resource

usage (demand string) is plotted as a function of the virtual time of that user.



Remember that the assumption was made that a program is a purely sequential process.

Observables of the system: Any quantities which are relevant to our study, among which might be included:

1) The effective progress rate w_i of job #i (working rate). It is the portion of time user #i was working, divided by the total real time interval over which the measurement was made.

$$w_i = \frac{\text{total virtual time interval for \#i}}{\text{total corresponding real time interval}}$$

2) The duty factor u_j of resource #j (or its proportion of usage):

$$u_j = \frac{\text{time resource \#j is used by and job}}{\text{total real time interval}}$$

w_i and u_j are both dimensionless variables, which are observed over a certain interval of real time.

3) The cost function (or economic criterion) of the system is another observable. It is assumed to be a weighted sum of the progress rates:

$$E = \sum_i c_i w_i$$

where c_i characterizes the urgency of user #i. The precise meaning of c_i as a bid will be discussed in section III.4.5.

III.3 A model based on fixed priorities (with preemption) for each user and resource.

III.3.1 Overview of the model.

For each resource, there is a priority assigned to each user. For a given resource, these priorities are all different (the users being totally ordered with respect to each resource). This priority assignment will not be changed during a certain time interval [0,T] over which the conflicts between the demand strings of the various users are studied. If user #i requests a resource, he will get it either if the resource is currently idle, or if it is allocated to a user having lower priority for the particular resource (in which case the lower priority user will have to wait for further use of this resource).

Note that a user does not necessarily have the same priority for all resources.

Let a_{ij} be the proportion of the virtual time of user i spent on resource j during the real-time interval [0,T]. The a_{ij}'s characterize the needs of the various users for the various resources (for instance, the degree to which they are compute-bound or I/O-bound). Given the virtual time diagram r(t) of user i, it is trivial to compute his a_{ij}'s:

$$a_{ij} = \frac{\int_{t \in [t_{min}, t_{max}]} \mathbb{1}\,(\,r(t)=j\,)\;dt}{t_{max} - t_{min}}$$

where $\mathbb{1}(x) = \begin{cases} 1 \text{ if } n = \underline{\text{true}} \\ 0 \text{ if } n = \underline{\text{false}} \end{cases}$

It will be assumed in the sequel that a fairly accurate knowledge of the a_{ij}'s is available (possibly based on past experience of the programs), but that the precise demand strings and virtual time diagrams of each program over the real-time interval [0,T] are not known.

The situation may be characterized by two matrices n x m, where n is the number of users and m the number of resources:

a_{ij} = proportion of the virtual time of user i spent on resource j

p_{ij} = integer number representing the priority of user i for resource j.

$1 \leq i \leq n$

$1 \leq j \leq m$

$0 \leq a_{ij} \leq 1$

$\sum_j a_{ij} = 1$ (normalization of the a_{ij} for each user)

$p_{ij} = p_{kj} \iff i = k$

$p_{ij} < p_{kj} \iff$ user i has a higher priority than k for resource j (lower numbers $\iff$ higher priorities).

The assumptions are given over a real-time interval [0,T], which separates the two activations of the macroscheduling algorithm. If w_i is the progress rate of user i, this user will effectively get resource j allocated during a time

$$T\, w_i\, a_{ij} \qquad \text{(by definition of } a_{ij} \text{ and } w_i\text{)}.$$

Resource j will be running during a total amount of time

$$T\, u_j = \sum_i T\, w_i\, a_{ij}$$

thus:

$$u_j = \sum_i a_{ij}\, w_i \qquad (0 \leq u_j \leq 1)$$

and, as a consequence of $\sum_j a_{ij} = 1$:

$$\sum_i w_i = \sum_j u_j$$

The overhead of switching a resource from one user to another has been and will be systematically neglected in the study of the model.

III.3.2 Fundamental equations and consequences.

It will now be of interest to derive the equations describing the worst possible cases, where the requests are synchronized in an order such as to get the smallest possible progress rates and the least possible simultaneous use of the resources available. This is relevant to the general philosophy that the system should always expect the highest amount of conflict within certain computed bounds. It should not oversell itself to the users, guaranteeing them a service that it would eventually not be able to give. Even if the system would decide to take some chances for a greater expected profit, probabilistic models would be dangerous because they assume a randomness and absence of correlation between users which are not generally true. Also, for a given user, the requests do not have a random length under some distribution, and are not uncorrelated with each other. Of course, the computer could compute Markovchain coefficients for the demand strings of the various users and use this information to get a better schedule, but this seems to exceed the allowable overhead of an allocator.

The following fundamental equations express that, in the "worst possible case", a process would be waiting for a resource at any time when this resource is used by another process of higher priority. Note that $1-w_i$ is the rate of waiting of user i.

$$(\text{III-1}) \qquad 1 - w_i \leqslant \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj}\, w_k \qquad \begin{matrix}(0 \leqslant w_i \leqslant 1) \\ (1 \leqslant i \leqslant n)\end{matrix}$$

For given matrices (a_{ij}) and (p_{ij}), it is always possible to find virtual time diagrams of the users, which have the usage ratios a_{ij} for the resources, and such that each job might have the maximum waiting rate given by equations III-1. In other words, if the coefficients a_{ij} are known for each job, but not the exact virtual time diagrams of the jobs, it can be said a priori that the jobs will have progress rates at least equal to the w_i's, if and only if the following equations are satisfied:

$$(\text{III-2}) \qquad 1 - w_i \geqslant \min \left(\sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj}\, w_k \;,\; 1 \right)$$

$$(\forall i:\ 1 \leqslant i \leqslant n)$$
$$(0 \leqslant w_i \leqslant 1)$$

An equivalent form is given in the following equations:

$$(\text{III-3}) \qquad \forall i,\ \underline{\text{either}}\ w_i = 0\ \ \underline{\text{or}}\ \ 1 \geqslant w_i + \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj}\, w_k$$

$$(w_i \geqslant 0)$$

Equations (III-2) define a domain of values for the w_i's. Any point within this domain can always be reached if the system should desire it.[8] This domain will be called the attainable domain, or

---

[8] The action to be taken by the system to reach a particular point in this solution space will be described in section III.4. A formal proof of this statement is not given here.

synonymously the domain of certainty. Note that equations (III-2) imply that:

(III-4) $$0 \leq u_j = \sum_i a_{ij} w_i \leq 1 \qquad (0 \leq w_i \leq 1) \quad (1 \leq j \leq m)$$

(The reader will find this result easy to prove).

III.3.3 Definition of the mathematical problem.

The previous model will be used to find a set of users to allocate. Each user gives to the system:

1. his usage ratios for the various preemptible resources: a_{ij}.[9]

2. his urgency: c_i. A higher value of c_i means a higher urgency, but also means that the user is willing to pay more money in order to run. It is understood that if a user with urgency c_i gets a progress rate w_i during a real time interval of length T, then this user is willing to pay at most

$$c_i w_i T$$

to run during this time interval. Pricing strategies are studied in III.4.5.

The mathematical programming problem can be stated in several forms of varying complexity.

1. Given the a_{ij}'s and the c_i's, find the p_{ij}'s and the w_i's which maximize the economic criterion (or cost function):

(III-5) $$E = \sum c_i w_i$$

while satisfying equations (III-2).

Note that the economic criterion chosen is equivalent to a cri-

---

[9] and, in the second formulation, his usage of the non-preemptible resources: b_{ij}.

terion which would tend to maximize resource usage. Suppose it is desired to maximize

$$E = \sum_j d_j u_j$$

where d_j is the "weight" or cost of resource #j. E can then be rewritten in the form:

$$E = \sum_j c_j w_i$$

with:

$$c_i = \sum_j a_{ij} d_j$$

2. In this second formulation, there are two kinds of resources: the macroresources, β (which are non-preemptible[10] and allocated by the macroscheduler), and the microresources, α (preemptible and allocated by the microscheduler). b_{ij} will be called the absolute amount of macroresource j desired by user i. By contrast, a_{ij} is the relative amount (per unit of virtual time) of microresource j needed by user i.

Now, the mathematical programming problem can be expressed in the following way:

Find a set of users S, and matrices (p_{ij}) and (w_i), which maximize the economic criterion (cost function):

(III-5) $$E = \sum_{i \in S} c_i w_i$$

subject to the following constraints:

---

[10] The non-preemptible resources might be, for instance, memory cells at various levels of memory (core, drum,...).

(III-6)

$$\begin{cases} \forall j \in \mathcal{B},\ \sum_{i \in S} b_{ij}\,\delta_i \leq B_j \\ \text{where } \begin{cases} \delta_i = 0 & \text{if } w_i = 0 \\ \delta_i = 1 & \text{if } w_i > 0 \end{cases} \\ \forall i \in S,\quad w_i + \sum_{\substack{j \in \mathcal{A} \\ p_{i',j} < p_{ij}}} a_{i',j}\, w_{i'} \leq 1 \end{cases}$$ [11]

B_j is the total available amount of non-preemptible resource #j.

III.3.4 Examples.

In this section are given a few examples of a graphical representation of the attainable domain in the space of the w_i's, in order to create a feeling of how the demand strings of the jobs will synchronize. The reader will discover that the "worst case domain" can nevertheless lead to a lot of simultaneity between the jobs. It was verified by simulation that the jobs do not generally synchronize significantly better than the worst case model predicts, if there is a small number of preemptible resources and if the priorities are chosen in a nearly optimal way (see section III.4.2).

Example 1.

$$(a_{ij}) = \begin{matrix} \text{CPU} & \text{DISK} \\ \begin{pmatrix} .8 & .2 \\ .4 & .6 \end{pmatrix} & \begin{matrix} \text{JOB 1} \\ \text{JOB 2} \end{matrix} \end{matrix}$$

[11] The equations for $j \in \mathcal{B}$ express that any user having a non-zero progress rate (and thus allocated by the macroscheduler, can find the space he needs in memory level #j.

In this array, a job is a row and a resource is a column. Resource 1 is the CPU, resource 2 is the disk; job 1 is compute bound, job 2 is more I/O bound.

1. The maximum priority is given to job 1 for all resources:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$$

Equations:

$$\begin{cases} 1 \geq w_1 \\ 1 \geq w_1 + w_2 \end{cases} \quad \text{or:} \quad \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The domain of certainty is defined by: $w_1 + w_2 \leq 1$, which shows that no parallelism in the use of the resources is obtained in the worst case (fig. III-1).

2. Suppose that the priority matrix is:

$$(p_{ij}) = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

Equations defining the attainable domain are:

$$\begin{cases} 1 \geq w_1 + .4\, w_2 & w_1 \geq 0 \\ 1 \geq .2\, w_1 + w_2 & w_2 \geq 0 \end{cases} \quad \text{or:} \quad \begin{pmatrix} 1 & .4 \\ .2 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

The attainable domain corresponds to a nearly optimal usage of the resources:

$$u_1 = .8\, w_1 + .4\, w_2$$

$$u_2 = .2\, w_1 + 16\, w_2$$

u_1 and u_2 are maximum at the point:

$$w_1 = .65,\ w_2 = .87 \rightarrow u_1 = .87,\ u_2 = .65$$

In this case, both u_1 and u_2 are maximum at the same point. This, however, is <u>not</u> a general result, and very complicated domains in the u_1 space might exist. Nevertheless, solving the equations:

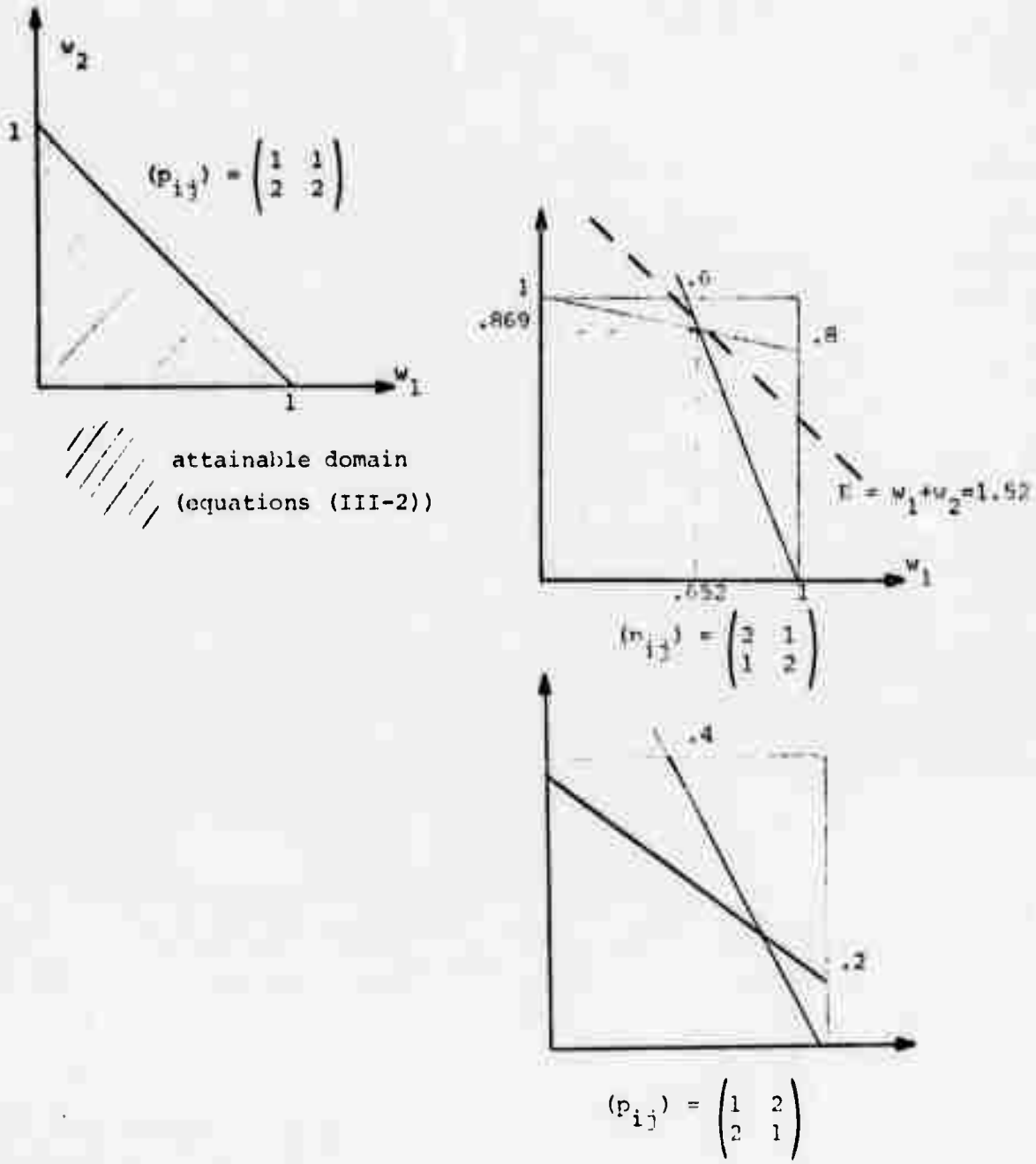


Fig. III-1

Domains in the (w_1, w_2) space for example 1.

$$\forall i, \quad 1 = w_i + \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj} w_k$$

might give a good approximation of the use of resources in the attainable domain.

3. The next case has the priority matrix:

$$(p_{ij}) = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$$

$$1 \geq w_1 + .6\, w_2$$

$$1 \geq .8\, w_1 + w_2$$

Obviously, the attainable domain is worse than with the second priority assignment, but better than with the first one.

If the objective function to be maximized is $E = w_1 + w_2$ ($c_1 = c_2 = 1$), then the priority assignment #1 yields $E_{max} = 1$, priority assignment #2 yields $E_{max} = 1.52$ and priority assignment #3 yields $E_{max} = 1.15$. The solution of the mathematical programming problem defined in the previous section would be the second priority assignment:

$$(p_{ij}) = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$$

and $w_1 = .65$, $w_2 = .87$.

Example 2:

This example considers two jobs having identical average resource usage characteristics, but one job seizes each resource during a much shorter amount of time than the other job.

Fig. III-2.1 and III-2.2 show the virtual time diagrams of each job. Fig. III-2.3 and III-2.4 show how they synchronize in real-time under two different priority assignments. The expected progress rates are found to be those given by the "worst case" model.

For this example:

$$(a_{ij}) = \begin{pmatrix} .5 & .5 \\ .5 & .5 \end{pmatrix}$$

$$(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix} \qquad 1 \geq w_1 + w_2$$

thus $E_{max} = w_1 + w_2 = 1 \quad (w_1 \simeq 1 \ , \ w_2 \simeq 0)$.

$$(p_{ij}) = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} \Rightarrow \begin{pmatrix} 1 & .5 \\ .5 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

thus $E_{max} = 1.33$ (with $w_1 = .67$, $w_2 = .67$).

Example 1 has a larger attainable domain and a larger E_{max} than example 2 with the priority assignment $\begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}$. This is due to the fact that the jobs of example 1 are complementing each other (one needs more CPU, the other more I/O), while jobs of example 2 have identical average needs of resources.

III.3.5 Multiprocessor case.

So far only the case where the resources are not interchangeable, and are only susceptible to one activation at a time has been considered. How the previous model can be extended to the case where some resources may have more than one activation at a time will now be studied. For instance, there might be several identical CPU's or identical channels.

The fundamental "worst case" equations are quite complicated. They are given here without further justification.

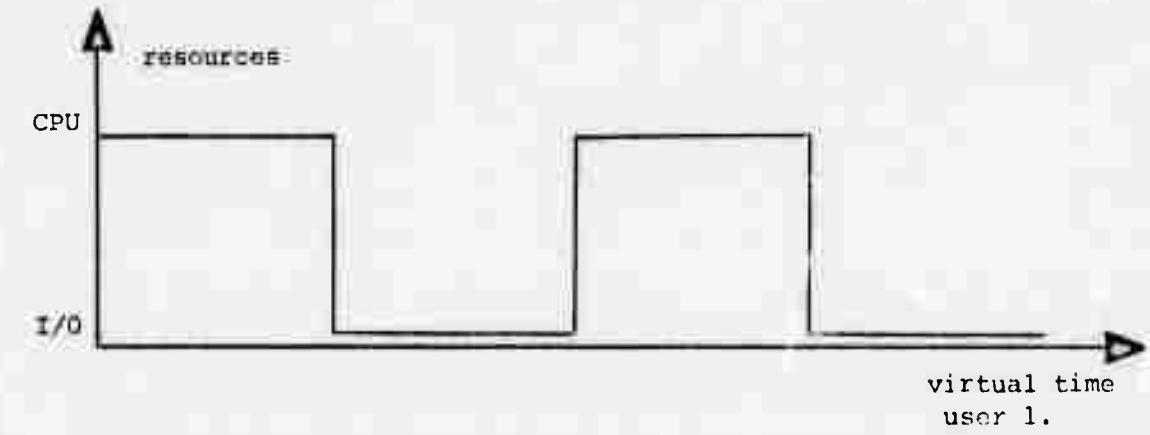


Fig. III-2.1: Virtual time diagram of user 1.

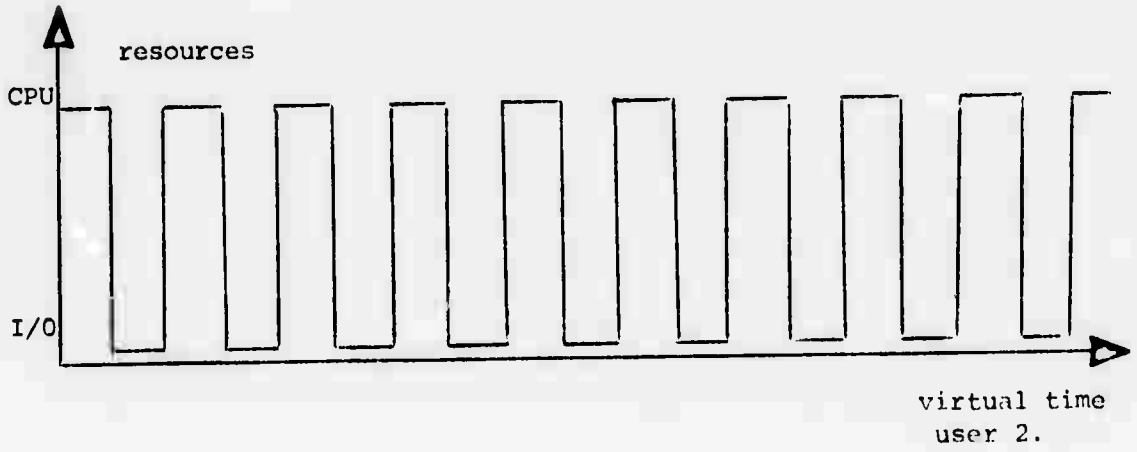


Fig. III-2.2: Virtual time diagram of user 2.

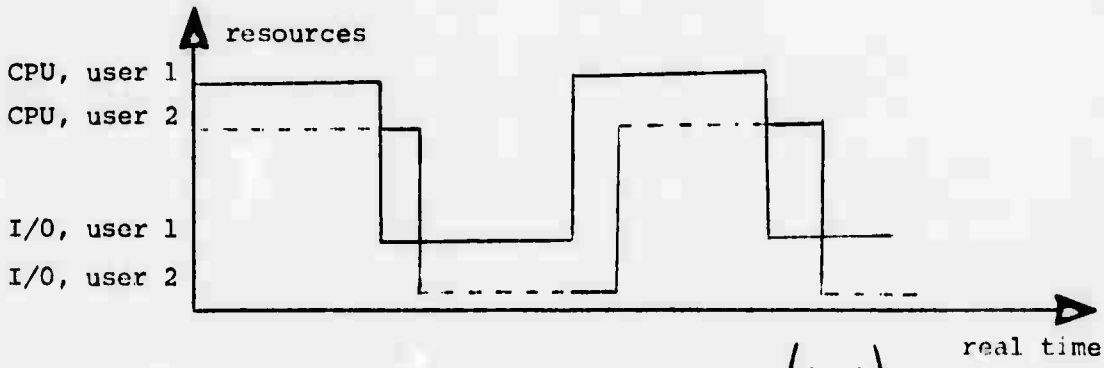


Fig. III-2.3: Real time diagram: $(p_{ij}) = \begin{pmatrix} 1 & 1 \\ 2 & 2 \end{pmatrix}$

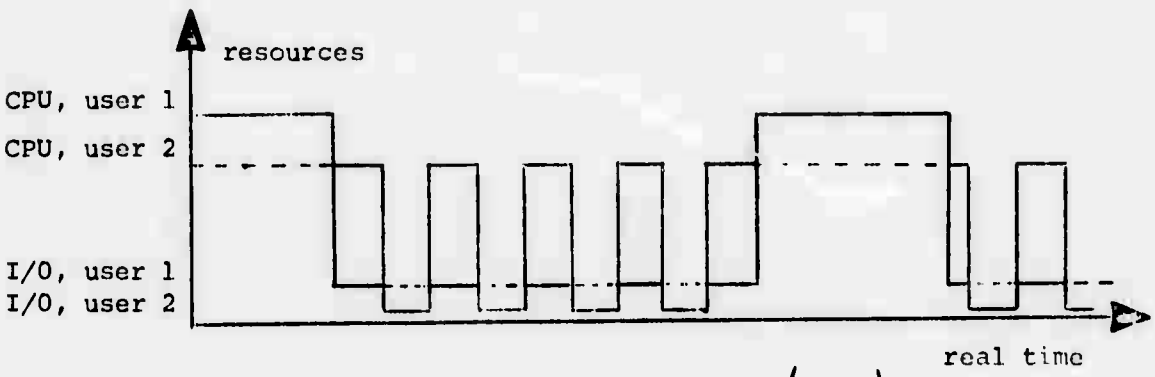


Fig. III-2.4: Real time diagram; $(p_{ij}) = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix}$

------ user waiting for the resource

_____ users seizing the resource

Define q_{rj} by: $r = p_{ij} \Leftrightarrow i = q_{rj}$ $\quad r \in [1,n]$, $j \in [1,m]$

q_{rj} is the number of the user having the r-th priority for resource j.

If resource j has R_j processors (possible simultaneous activations), the maximum time that user i would spend waiting for resource j in time interval [0,T] is:

$$\eta_{ij} = \begin{cases} \min\limits_{k \in [1,R_j]} \left(\dfrac{1}{R_j - k + 1} \sum\limits_{m \in [k,r-1]} a_{q_{mj}j} \; w_{q_{mj}} \right) & \text{if } r = q_{ij} > R_j \\ 0 \quad \text{if } q_{ij} \leq R_j \end{cases}$$

so that the fundamental equations for the attainable domain are:

$$\forall i \in [1,n] \quad \underline{\text{either}} \quad 1 - w_i \sum_{j \in [1,m]} \eta_{ij} \quad \underline{\text{or}} \quad w_i = 0 \qquad (\text{always } w_i \geq 0)$$

Theorem 1. A smaller domain than the attainable domain defined by the previous equations can be defined by equations (III-2) where a_{ij} has been replaced by $\alpha_{ij} = \dfrac{a_{ij}}{R_j}$.

Proof: by choosing the first of the quantities whose minimum is η_{ij}:

$$\eta_{ij} \leq \frac{1}{R_j} \sum_{i \in [1,r-1]} a_{ij} \; w_i \qquad \text{if } r = q_{ij} > R_j$$

and $$\eta_{ij} = 0 \leq \frac{1}{R_j} \sum_{i \in [1,r-1]} a_{ij} \; w_i \qquad \text{if } r = q_{ij} \leq R_j$$

hence the theorem.

By replacing the a_{ij}'s by the α's the multiprocessor case has been reduced to a monoprocessor case. The accuracy of this approximation can be felt even more in the special, but very important case where $a_{ij} < a_{kj} \Leftrightarrow p_{ij} < p_{kj}$. In that case, if $r = q_{ij} > R_j$:

$$\eta_{ij} = \frac{1}{R_j} \sum_{i \in [1,r-1]} a_{ij} w_i$$

and, for $r = q_{ij} \leq R_j$, $\eta_{ij} = 0$, but then the a_{ij}'s are small too, because they yield high priorities. This leads to the intuitive feeling that for this priority assignment and a large number of users ($n >> R_j$), the exact model and the model approximated by the α_{ij}'s are very similar to each other.

There is another way of approximating the multiprocessor case by a monoprocessor case. Suppose resource j to be a single resource (one activation only), but which works at R_j times its initial speed. The fundamental equations would then be:

$$1 - w'_i \geqslant \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} \beta'_{kj} w'_k$$

where β'_{kj} is the normalized value of $\frac{a_{ij}}{R_j}$:

$$\beta'_{kj} = \frac{\frac{a_{ij}}{R_j}}{\sum_{j'} \frac{a_{ij'}}{R_{j'}}}$$

The progress rates w'_i apply in a universe where the resources work faster than initially. The correspondence between w'_i and w_i is given by:

$$w'_i = w_i \sum_j \frac{a_{ij}}{R_j}$$

Thus, the fundamental equations for the processors working at R_j times their initial speed are:

$$\forall\, i\,, \quad 1 - w_i \left(\sum_j \frac{a_{ij}}{R_j} \right) \geqslant \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} \frac{a_{kj}}{R_j}\, w_k$$

The reader can see that this is a more optimistic estimate than was given by theorem 1 and the α_{ij}'s.

Example 3.

a_{ij}	CPU	Bus
job 1	.8	.2
job 2	.4	.6
job 3	.2	.8

Suppose that 2 CPU's are available. The following priorities are assigned.

$$(p_{ij}) = \begin{pmatrix} 3 & 1 \\ 2 & 2 \\ 1 & 3 \end{pmatrix}$$

and $E = w_1 + w_2 + w_3$ is to be maximized.

1. An exact treatment gives the fundamental equations:

$$1 \geq w_1 + \min\ (.2\ w_2 + .1\ w_3)$$

$$1 \geq .2\ w_1 + w_2$$

$$1 \geq .2\ w_1 + .6\ w_2 + w_3$$

yielding the solution (with equalities):

$$\begin{cases} w_1 = .94 & w_2 = .81 & w_3 = .32 \\ \frac{1}{2}\ u_1 = .57 & u_2 = .93 \end{cases}$$

$$E_{max} = 2.07$$

2. The lower bound method:

$\alpha_{ij} = \frac{a_{ij}}{R_j}$	CPU	Bus
job 1	.4	.2
job 2	.2	.6
job 3	.1	.8

$$\begin{pmatrix} 1 & .2 & .1 \\ .2 & 1 & .1 \\ .2 & .6 & 1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

$$\left.\begin{array}{l} w_1 = .80 \quad w_2 = .80 \quad w_3 = .36 \\ \frac{1}{2}\ u_1 = .52 \quad u_2 = .93 \end{array}\right\} \text{ is the "best" point of the worst case.}$$

$$E_{max} = 1.96$$

3. The "optimistic" approximation with a CPU working twice as fast:

$$\begin{pmatrix} .6 & .2 & .1 \\ .2 & .8 & .1 \\ .2 & .6 & .9 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix} \leq \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}$$

thus:

$$w_1 = 1.33 \quad w_2 = .88 \quad w_3 = .21$$

$$E_{max} = 2.44$$

Note that w_1 is greater than 1, which is not surprising with

the assumptions. If the constraint $w_i < 1$ is imposed, the optimum would be:

$$w_1 = 1 \quad w_2 = .97 \quad w_3 = .24$$

$$E_{max} = 2.21$$

which is a closer approximation of the case with two processors.

III.4 Macroscheduling and microscheduling algorithms under the previous model.

This section is intended to show how the model of section III.3 can be applied to multi-level scheduling. Section III.4.1 shows how the schedulers look from the users point of view. An attempt is then made to solve the mathematical programming problem by separating the assignment of priorities from the computation of the progress rates. A heuristic for assigning priorities is given and justified in III.4.2. Once priorities are chosen, the problem is reduced to a multi-dimensional Knapsack problem, for which a heuristical solution is proposed in III.4.3. Section III.4.4 summarizes the results, and III.4.5 shows how the users could be charged at "marginal prices"; the prices for services being determined in connection with the algorithm of III.4.3.3.

III.4.1 Combining two levels (in time) of scheduling.

When designing an operating system, one of the major difficulties is to partition the concepts involved. This requires, in particular, the separation of tasks which are loosely connected, and the implementation of them as separate processes of the system.[12] It is assumed

[12] This can be achieved with a hierarchical structure such as the one proposed by Dijkstra [26], with a communication system between processes [27].

here that the following decisions and strategies are independent from resource allocation (or scheduling), except that they may provide information or requests for the scheduler, but they should not be confused with scheduling activities:

1. Page replacement algorithms in a computer with paged memory. Which page should be extracted from memory? Should the size of the working set of pages be changed? Should there be any prepaging?

These decisions can be made by the programmer, or by the system, but in any case they concern program optimization and not system optimization (insofar as it can be said that the system is not improved by improving the users' programs running under it).

2. Deciding on the "external" priorities of jobs. Some jobs are more urgent than others. This might be decided either by the system or by the user (which is willing to pay more to get his job executed soon). External priority can be reduced to the economic criterion of the price offered by the user per unit computation of his job, which is then fed into the scheduler, and will serve the scheduler in order to build its own optimization criterion.

3. Statistics to be used by the scheduler or by the paging algorithm or to compute external priorities, can theoretically be considered to be collected independently of these decision-making processes.[13]

A scheduler will be considered to be a mechanism using the following information:

---

[13] If the user wants to collect such statistics, he will have to pay for the resources involved in the spying process.

1. On each program: some information about the kind of service wanted by the program, either on a long term range, or because of a current request for a facility. The a_{ij}'s of section III.3 were an example of such long term range information.

2. The economic "bid" of each job, characterizing its "external" urgency.

3. The resources available to the system.

I believe that some scheduling should be done for intervals of various duration of time. For instance, microscheduling, as defined in the sequel, might be done for periods in milliseconds, macroscheduling for periods in hundreds of milliseconds, while scheduling of tapes should be done for minutes, and some real time users may not want to use the system at all unless they are assured of getting some minimum guaranteed resource usage during a whole hour.

A scheduler decides to allocate some resources to some users, and chooses parameters to be fed into the "lower level" scheduler (which handles smaller time intervals).

Microscheduling and Macroscheduling.

Examples will now be given of how some usual allocation decisions can be split between two schedulers working at different levels of time intervals. These examples are summarized in table III-1. Resources allocated by the low-level scheduler (microscheduler) are said to be preemptible, and those allocated by the macroscheduler are said to be non-preemptible.[14]

---

[14] Preemptibility is, of course, a relative notion (and not absolute). It can just be stated that some resources are more preemptible than others, if the overhead to allocate them is smaller.

	Current Computer Systems	Future Computer Systems
Macroscheduling	Allocation of core memory	Allocation of core memory and fast registers
Time between macro-decisions	≈ 1 sec.	≈ 10 millisec.
Microscheduling	Conflicts of accesses to drum and disk Allocation of CPU and fast registers	Allocation of CPU conflicts in swapping between central core and fast registers
Time between microdecisions	≈ 10 millisec.	≈ 100 microseconds

Table III-1

Examples of macro and micro scheduling

Macroscheduling can refer to those scheduling operations which are related to the allocation of central memory. The time interval between two macrodecisions would be rather large (greater than 100 milliseconds on most systems). Microscheduling would concern the allocation of the arithmetic and control units and of some fast busses, to programs which are already essentially present in central memory. For instance, the decision of what job is allocated access to the drum for page-in and page-out operations between drum and main core memory is a microscheduling decision in current computer systems where the programs are kept in core while paging takes place. In future computer systems, this kind of paging will most probably be replaced by a paging between two fast levels of memory, like on the 360/85.

I prefer the words "microscheduling" and "macroscheduling" to "microqueuing" and "macroqueuing" [5], because the latter suggest the use of FIFO queues by the scheduling algorithm, which is a practice that this chapter precisely tries to discredit. Note that, in our terminology, "scheduling" and "allocating" are synonymous.

The macroscheduler receives the predicted probable usage ratios, a_{ij}, and the urgency, c_i, for each user #i who wants to run. Then the macroscheduler will solve the Mathematical Programming problem (defined in III.3.3). Having determined the set of users to be allocated during a certain real-time interval of length T, and the matrices (p_{ij}) and (w_i) for the users in this set, the macroscheduler assures that the non-preemptible resources will be allocated to those users, and transfers the values of (p_{ij}) and (w_i) to the microscheduler.

The microscheduler controls the access of the users to the pre-

emptible resources, by applying the priorities which were determined by the macroscheduler. It also stops a user #i if this user runs more than a time $w_i T$. Finally, it prevents the users from exceeding their predicted resource usage ratios.[15]

Sections III.4.2 and III.4.3 will be devoted to the solution of the M.P. problem by the macroscheduler.

III.4.2 Assignment of priorities.

It is interesting to try to find a priority assignment (p_{ij}) before determining the progress rates (w_i) which optimize the cost function $E = \Sigma\, c_i\, w_i$.

III.4.2.1 A case where a given assignment of priorities can certainly be improved.

Let D_p be called the attainable domain under priority assignment P. It will now be shown that the assignment to each job of a set of the same priorities for all resources is a wrong choice, which can always be improved. Consider the following theorem:

Theorem 2: Let P be a priority assignment in which $p_{ij} < p_{i'j}$ for all j. If, for some j, $p_{i'j} = p_{ij} + 1$, then there exists another assignment P' which is the same as P except that p_{ij} and $p_{i'j}$ are interchanged, such that $D_{p'}$ properly contains D_p.

In other words, any point in the space of observables which satisfies equations (III-2) under assignment P, will satisfy (III-2) under priority assignment P'.

---

[15] See section III.4.4 for a precise sketch of the microscheduling algorithm.

Corollary: the maximum of the cost function[16] under priority assignment P' will be greater than or equal to its maximum under P.

The proof of the theorem is given in appendix A. The theorem is quite weak, but at least it shows that a priority assignment like the following can certainly be improved:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 3 & 3 \\ 4 & 4 & 4 \end{pmatrix}$$

III.4.2.2 Two cases where it is known how to assign priorities.

1. The users are competing on one resource only.

There might be more preemptible resources in the system, but for each of the other resources there is no more than one user who might ask for it.

In this case, the fundamental equations take the form:

$$\forall i \in [1..n], \quad \text{either } w_i = 0 \text{ or } 1 - w_i \geqslant \sum_{\substack{k \\ p_{kj} < p_{ij}}} a_{kj} w_k \qquad (0 \leqslant w_i \leqslant 1)$$

where j is the critical resource, and:

$$E = \sum_i c_i w_i$$

is to be maximized.

Theorem 3: The optimal priority assignment, P, is such that:

$$p_{ij} < p_{kj} \quad \Leftrightarrow \quad \frac{a_{ij}}{c_i} < \frac{a_{kj}}{c_k}$$

The proof is sketched in appendix B.

---

[16] or economic criterion.

2. <u>There is a finite number of users and resources, but the a_{ij}'s are infinitely small (of the first order)</u>.

Under these assumptions, $1-w_i$ is a first order, infinitely small number. Thus:

$$1 - w_i \sim \sum_{\substack{j,k \\ p_{kj} < p_{ij}}} a_{kj}$$

and:

$$n - E_{max} \sim \sum_{\substack{i,j,k \\ p_{kj} < p_{ij}}} c_i \, a_{kj}$$

The question of which priority system makes $n-E_{max}$ the smallest possible, still exists. The following theorem solves the problem:

<u>Theorem 4</u>: The optimal priority assignment P is such that, for any resource j and any users i and i',

$$p_{ij} < p_{i'j} \quad \Leftrightarrow \quad \frac{a_{ij}}{c_i} < \frac{a_{i'j}}{c_{i'}}$$

Proof:

$$n - E_{max} \sim \sum_{j} \sum_{\substack{i,k \\ p_{kj} < p_{ij}}} c_i \, a_{kj}$$

Thus, the problem of finding the optimal priorities can, in this case, be solved for each resource independently from the values of the a_{ij}'s and p_{ij}'s for the other resources. Theorem 3 can then be applied, and gives the optimum priority assignment for each resource.

III.4.2.3 General case.

The general problem of assigning priorities is generally quite complicated. Using the results of sections III.4.2.1 and III.4.2.2, suggests the following heuristic:

1. Assign priorities so that:

$$p_{ij} < p_{kj} \quad \Leftrightarrow \quad \frac{a_{ij}}{c_i} < \frac{a_{kj}}{c_k}$$

2. Try to improve this priority assignment by using theorem 2. This improvement can be achieved in a time proportional to m x n.

This priority assignment is not always optimal, as shown by the following counter-example:

Example 4: 3 resources, 2 jobs.

$$(a_{ij}) = \begin{pmatrix} .3 & .3 & .4 \\ .31 & .6 & .09 \end{pmatrix}$$

It is desired to optimize $w_1 + w_2$. The preceding method leads to the following assignment of priorities:

$$(p_{ij}) = \begin{pmatrix} 1 & 1 & 2 \\ 2 & 2 & 1 \end{pmatrix}$$

The optimum is $w_1 + w_2 = 1.384$. However, with the priority assignment:

$$(p_{ij}) = \begin{pmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

the optimum would be: $w_1 + w_2 = 1.477$.

This example clearly shows that the "good" priority assignment is not always optimal. The major advantage of this method of assigning priorities is simplicity.

III.4.3 Assignment of values for the progress rates w_i.

Once the p_{ij}'s have been determined, it is desirable to determine optimum values for the progress rates in order to maximize E while satisfying equations (III-2).

III.4.3.1 The 0/1 integer linear programming problem.

Now the mathematical programming problem which was defined in part 2 of section III.3.3 will be considered: maximize (III-5) subject to the constraints (III-6) (except that the p_{ij}'s have already been determined).

This mathematical programming problem is not a 0/1 integer linear programming problem, but it is convenient to consider it as such (determine the values of the δ_i's equal to 0 or 1). Note that the a_{ij}'s and the b_{ij}'s are all positive (they represent the needs of the users for preemptible and non-preemptible resources).

The 0/1 integer linear programming problem has been reviewed in [23]. Of particular interest to us are the studies in [24,28]. The idea is to find a nearly optimal set of users to be allocated (δ_i = 0 or 1) by ordering the users according to some criterion which will be called "decreasing desirabilities", and to try to allocate them (satisfy the constraints), starting with the user having the highest desirability.

III.4.3.2 A first algorithm.

The jobs are supposed to be already ordered by decreasing external priorities. It is then necessary to decide about the w_i's for the jobs. For instance, suppose $w_i = .5$ for the high priority jobs. A w_i too close to 1 might strongly degrade the possible service for other jobs, by obliging the system to give a high internal priority for all resources to the job which has a high external priority. This would lead, as has been seen, to a poor utilization of the resources.

If a set S of jobs is allocated in memory, it has to satisfy equations (III-3) and (III-6) (with $\delta_i = 1$ if $i \in S$, $\delta_i = 0$ otherwise).

A procedure to find the maximal set of users fitting into available resources would take the following steps:

Step 1: Take the highest priority user; put him in set S'.

Step 2: Check whether S' is an allowable set: first assign the priorities p_{ij} $\forall i \in S'$, according to the rule

$$p_{ij} < p_{kj} \iff \frac{a_{ij}}{c_i} < \frac{a_{kj}}{c_k}$$

where the c_i's are in the same order as the external priorities. Then check equations (III-3, and III-6) for set S'. If they all check, go to step 3, else go to step 4.

Step 3: $S \leftarrow S'$; go to step 4.

Step 4: Define S' as including all users of S, plus the highest priority user not yet handled. If there are no more users to handle, the algorithm stops, else go to step 1.

Using the above procedure a maximum allowable set of users has been found, each of which has a requested guaranteed service. The

computations can be done so that the time required by the algorithm is: $t = A\ m\ n + B\ m\ n \log(n)$. The $n\log(n)$ term expressed the time to sort the quantities $\frac{a_{ij}}{c_i}$ (to determine the priorities).

Set of users with non-guaranteed service.

Assume that the non-preemptible resources are not saturated after having applied the previous macroscheduling algorithm. Some other users might then be allocated with priorities lower (for each preemptible resource) than the lowest priority of the users of set S. The guaranteed service of users of set S will not be affected by these additional ("Marginal") users. M will be the set of marginal users.

Priorities in set M are determined according to the same criteria as in set S. Of course, the resource usage will not be as good as if the priorities had been determined optimally for the entire set M + S. Our solution respects the external priorities of the users, while maximizing the system's efficiency.

Example 5: There are 2 CPU's but only one bus (or channel).

a_{ij}	CPU	I/O	w_i decided if allocated
job 1	.4	.6	.5
job 2	.3	.7	.5
job 3	.5	.5	.5
job 4	.9	.1	.5
job 5	.8	.2	.5
job 6	.6	.4	.5

These 6 candidates are in the order of their external priorities. There is no constraint due to non-preemptible resources in this example.

The reader can verify that the algorithm (with $c_i = 1, \forall i$) will accept jobs 1 and 2, reject 3, and accept 4 and 5. This is intuitively

a good choice because 1, 2, 3 are I/O bound while the others are compute bound. $S = \{1,2,4,5\}$, $M = \{3,6\}$, and the priority assignments are:

$$(p_{ij}) = \begin{pmatrix} 2 & 3 \\ 1 & 4 \\ 5 & 6 \\ 4 & 1 \\ 3 & 2 \\ 6 & 5 \end{pmatrix}$$

Note that job 3 would have been accepted if $w_3 \leq .4$. This assignment gives the resource usage $u_{CPU} = .6$, $u_{I/O} = .8$, which could be improved by solving equations (III-2) (equalities) for the w_i's with the p_{ij}'s that were just computed.

Comments:

1) When deciding about the desirabilities of the jobs, only the external priorities and not a more precise quantitative measure of their urgencies were taken into account.

2) Clearly, if the w_i's were computed, instead of just being arbitrarily decided before the algorithm started, a more optimal solution could have been obtained.

III.4.3.3 A more general algorithm.

The following algorithm attempts to find a nearly optimal solution to the problem. It works in two steps:

1) Get an approximate solution by optimizing the economic criterion (III-5) with the following constraints:

$$\text{(III-7)} \quad \begin{cases} \forall\, j \in \mathcal{B} & \displaystyle\sum_{i \in S} \frac{b_{ij}}{B_j} \leqslant 1 \\ \forall\, j \in \mathcal{A} & \displaystyle\sum_{i \in S} a_{ij}\, w_i \leqslant 1 \end{cases}$$

The constraints for the resources of set $\mathcal{B}$ are identical in equations (III-6) and (III-7). The constraints concerning the resources of set $\mathcal{A}$ are, however, weaker in equations (III-6) than in equations (III-7). The latter just express the best possible case (where no unnecessary interference between jobs would happen), however, this method is used because equations (III-7) are easier to manipulate than equations (III-6) and a more refined solution will be attained later. This first step is essentially intended to eliminate from further consideration the jobs which should certainly not be scheduled (for which $w_i = 0$).

To get a good approximate solution of this mathematical programming problem (III-7), it is not necessary to use an enumerative method of search. A faster method which gives a good approximate solution works as follows:

Assign an initial weight K_j to resource j.[17] Assign an initial w_i to job i. Compute the desirability for each job:

$$d_i = \frac{c_i}{\sum_{j \in \mathcal{B}} b_{ij} K_j + \sum_{j \in \mathcal{A}} a_{ij} w_i K_j}$$

Sort the jobs according to their desirabilities. Starting with the one of highest desirability, compute whether the job can be allocated or not, that is, if equations (III-7) can be satisfied with S consisting of the jobs which have already been allocated and of the job which is a candidate to be added. Whether the job has been allocated or not, try the next one.

---

[17] The initial weights when the microscheduler is activated might be the final weights obtained at its previous activation.

When all the jobs have been examined, compute a new weight assignment (the K_j's) and the new w_i's according to the principle that a job having larger d_i should have a larger w_i, and that a resource for which the corresponding equation (III-7) had its left side much smaller than 1, should have its weight decreased.

This entire process can be repeated 2 or 3 times.[18]

2) Having determined the set S, a better approximation of the w_i's can be determined by solving equations (III-8), with

$p_{ij} < p_{i'j} \iff a_{ij}/c_i < a_{i'j}/c_{i'}$.

$$1 = w_i + \sum_{\substack{j \in a \\ p_{i'j} < p_{ij}}} a_{i'j}\, w_{i'} \qquad \forall\, i \in S \tag{III-8}$$

If any of the w_i's of the solution is negative, this w_i is removed from set S, and equations (III-8) are solved again. As shown, equations (III-7) gave a set of users to be allocated which could be somewhat too large. Eliminating some users from this set in some cases, yields a nearly optimal set to satisfy equations (III-6) while maximizing E given by equation (III-5).[18]

III.4.4 A summary of the proposed scheduling method.

1) Macroscheduling: It has been shown how, given the a_{ij}'s, b_{ij}'s and c_i's, the macroscheduler determines the p_{ij}'s and w_i's and

---

[18] This algorithm has been programmed and checked for several examples, for various numbers of users and resources. It always worked satisfactorily. Note that the choice of the rule used to get a new set of weights is essential to obtain a fast convergence.

transmits them to the microscheduler. It also allocates the non-preemptible resources for a period of time T.

2) Microscheduling: the microscheduler keeps track of the usage of preemptible resources by the allocated users. If user i uses resource j during more than a time

$$w_i a_{ij} T$$

then job i is punished in the sense that its priority p_{ij} for this resource is changed to a priority lower than any job which had not exceeded its quantum on the resource. This method assures that a job which accurately estimated its needs will be served at least as well as promised.

This changing of priorities by the microscheduler does not affect the previsions of the mathematical model (which assumed that the microscheduler did not touch the priorities but only inforce them), for priorities are only changed when a user exceeds his allowed quantum on a resource.

III.4.5 Pricing.

The determination of prices is, to a large extent, a consequence of the scheduling strategy. In the approach taken, a user agreed to pay at most a price $c_i w_i$ to get a progress rate w_i, and if he proposed a larger c_i he got higher priority.

However, the system should charge the various jobs being allocated more or less uniformly. It should not just charge the maximum possible to each job, because otherwise the jobs would start with very low c_i's and then increase them slowly until they were scheduled, thus leading to a greater overhead. The marginal theory of pricing theoret-

ically requires the system to charge user i exactly $c'_i w_i$, where $c'_i \leq c_i$ is the lowest bid that the user would have had to offer to get allocated. Unfortunately, this definition would lead to very complicated computations. I suggest here a few alternative methods.

1) If ℓ is the first job which was skipped (not allocated) when the jobs were scanned in order of decreasing desirabilities in the first step of the macroscheduling algorithm, and if w_i the effective progress rate for job i, charge job i:

$$p_i = \min (c_\ell , c_i) \times w_i \times T$$

2) If jobs having estimated their a_{ij}'s incorrectly are to be penalized, and if job i has effectively used an amount r_{ij} of resource j, he is charged:

$$p_i = \min (c_\ell , c_i) \times \max \left(\frac{r_{ij}}{a_{ij}}\right)$$

3) A unit cost for resource j could also have been computed:

$\mu_j = K_j d_\ell$, where ℓ is the first job not allocated and K_j the weight of resource j, as computed by the macroscheduling algorithm. If job i uses resource j during a time r_{ij}, he could be charged:

$$p_i = \min \left(c_i w_i , \sum_j r_{ij} \mu_j \right)$$

It is useful to have some prices for resources, so that:

1) A new coming user can by immediate inspection of the prices determine whether he wants to get on the system or not.

2) On the long range, the computer center staff might determine the needs to install or remove facilities (see Nielsen [16]).

The variations of the μ_j's in time should probably be smoothed

for those purposes.

III.5 Models of other priority systems.

III.5.1 The equipriority case without preemption.

In this "no-priority case", a user seizing a resource will never be preempted and will not lose the resource until he decides to release it. The situation may lead to almost no parallelism in the computations.

The worst case equations are:

$$(III\text{-}9) \qquad \forall i \in [1,n], \qquad 1 \geq w_i + \sum_{\substack{k \neq i \\ j}} a_{kj} w_k$$

In the situation of example #3, this gives the following progress rates:

$$w_1 = 16,\ w_2 = .32,\ w_3 = .64$$

so that the overlap of activity is small:

$$\text{overlap} = w_1 + w_2 + w_3 - 1 = 12\%$$

Theorem 5: The attainable domain D_N of the no-priority system is properly contained in the attainable domain D_p of any priority system.

Proof: compare equations (III-2) and (III-9). The latter imply the former.

Therefore the no-priority case is uninteresting, and should be avoided in any actual system design.

III.5.2 The equipriority case with preemption.

This case would also be called the case of "Randomly turning" priorities. The model is characterized by the following microscheduling method:

The time is divided into very short intervals, and the priority of

the users for the various resources is changing from one interval to the other, cycling so that each user spends the same amount of time in each priority level. Typically, the time between two priority changes might be 100 microseconds and is small compared to the interval between two allocation requests of jobs to the microscheduler. Nevertheless, assume that this method does not introduce any additional overhead.

A random number generator might be used at the beginning of each time interval, to generate the job priorities during this interval. This would insure that there is no regular pattern of one job spending most of the time at a higher priority than another, as happens with a circular permutation.

The idea of such a microscheduling algorithm has the following justifications:

1) The hardware could allow time-sharing of a CPU or a channel on very short time-slices, however, we don't know whether this would be a good practice.

2) It is desirable to assure a user of a certain percentage of use of some resources, under any circumstances. Time-slicing on a very short time basis might seem a natural way to do it. If user i is assured of having the top priority on resource j during a portion of time $\lambda_j \Delta T$ where ΔT is some small interval of time, then, with the a_{ij}'s defined previously, his progress rate will be at least

$$w_i = \min_j \frac{\lambda_j}{a_{ij}}$$

However, a much higher "lower bound" estimate for the w_i's can be computed. After having done it, these new "worst case" equations will be compared to equations (III-2) and it will be shown that, under some

assumptions, the "turning priorities microscheduling" performs poorer than a fixed prior[illegible] algorithm with the p_{ij}'s well chosen. This result has been checked by simulation, and the following discussion attempts to establish a theoretical justification.

Under this new model, if k users compete for some resource, each one will get it during a portion of the time 1/k. Consider resource j. User i will seize it during a period $T\ a_{ij}w_i$. In the worst possible case, the maximum overlap of requests occurs on resource j. Thus, the time spent by user i waiting for resource j is less than or equal to

$$\sum_{k \neq i} \min\ (a_{kj}w_k\ T,\ a_{ij}\ w_i\ T)$$

This points out that if a job k asks for less time on resource j than job i, the maximum time spent by job i waiting for resource j because of job k will be $T\ a_{kj}w_k$. If, on the other hand, $a_{ij}w_iT<a_{kj}w_kT$, job i will wait for resource j because of job k at most during a time $a_{ij}w_iT$. (see fig. III-3).

The worst case equations are thus:

(III-10) $$1 \geqslant w_i + \sum_{\substack{k\neq i \\ j}} \min\ (\ a_{kj}\ w_k\ ,\ a_{ij}\ w_i\ ) \qquad \forall i \in [1,n] \quad (w_i \geqslant 0)$$

These equations define the attainable domain with turning priorities.

Theorem 6: For every point in the attainable domain defined by equations (III-10), there exists a priority system in which this point is attainable according to equations (III-2).

Proof: Define this priority system by:

$$p_{ij} < p_{kj} \quad <=> \quad a_{ij}\ w_i < a_{kj}\ w_k$$

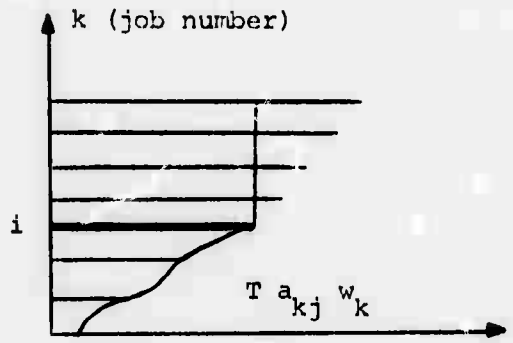


Fig. III-3: Time spent by the jobs on resource j and maximum interference of job i with other jobs.

Assume that, for a given j, the a_{ij}'s are all different. Then obviously, equations (III-10) imply equations (III-2) for this system, which are:

$$\forall i \in [1,n] \qquad 1 \geqslant w_i + \sum_{k \neq i} a_{kj} w_k$$

$$a_{kj} w_k < a_{ij} w_i$$

This theorem is reassuring because it says that whatever a user is assured of doing under a turning priority system, he is also assured of doing under a fixed priority system.

However, the following theorem can be proved under some restrictive assumptions:

Theorem 7:

If one of the following is true:

1) There are only 2 jobs (and any number of resources).

2) There is any number of jobs, but competition is limited to one resource only; then there exists a priority system whose attainable domain includes the domain defined by:

$$p_{ij} < p_{kj} \quad <=> \quad a_{ij} < a_{kj}$$

The proof is shown in appendix B.

Theorems 6 and 7 show that a fixed priority system should, to a certain extent, be preferred to a random priority system (which is itself better than no preemptibility at all). If a resource has the property that it can be preempted without any other additional future loss of time, then the available information on the jobs can be used to assign priorities for the resources, and a "good" choice is to assign the resource to the job which has the least need for it (after having

weighted these needs by the external urgencies of the jobs, which leads to the quantities $\frac{a_{ij}}{c_i}$.

III.6 Problems for further research.

1) Continuous macroscheduling: Instead of applying the macroalgorithm at regular time intervals, find a simplified macroalgorithm to be applied each time a job previously running deactivates itself voluntarily, or when a job changes its external priority, or even when the swapping channel is idle. Jobs might be scheduled or unscheduled just using the desirabilities which have already been computed, but it might also be desirable to recompute the p_{ij}'s, the K_j's, the d_i's and the w_i's.

2) Extend the models to include processes using more than one resource at a time. For instance, Fig. III-4 shows the virtual time diagram of a user who initiates I/O and swapping at the same time:

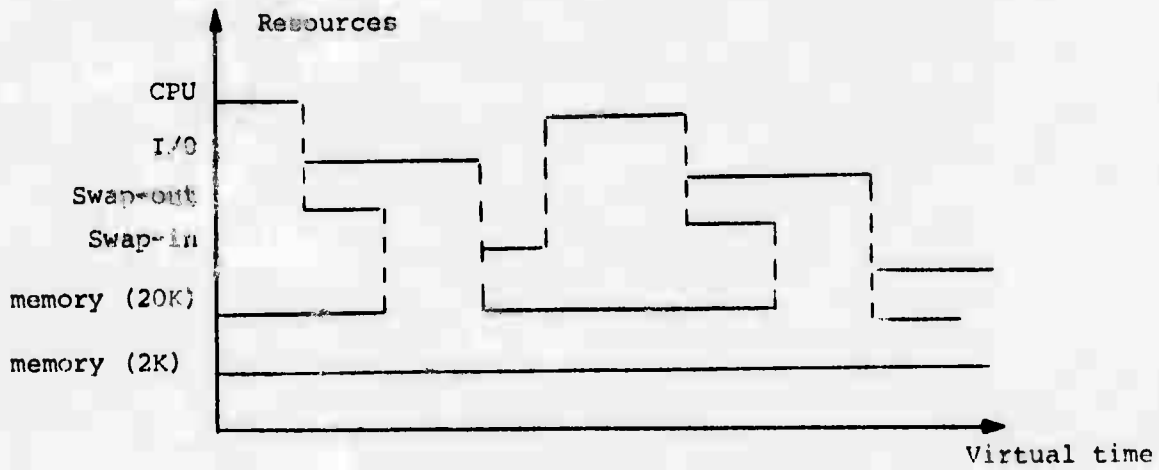


Figure III-4

Another characteristic of our hypothetical job is that it does not need all its memory resource continuously (a buffer of 2K is enough

during I/O completion). Could this knowledge be taken care of?

Solving this problem would be especially useful for future computer systems where the cost of arithmetic and control units is expected to decrease much more than the cost of central memories.

3) Find models of "probable" performance as well as "worst case" models.

4) Which information other than a_{ij}'s or the b_{ij}'s on the jobs would be relevant to an allocation algorithm?

For instance, the exact virtual time at which a job will place a request might be available for some jobs while being completely out of the question for others.

5) How much would the results of the model be affected by slight errors in the predictions?

III.7 Conclusion.

My initial effort was applied to separate problems which are usually handled together in a very intricate manner: 1) Scheduling; 2) Paging algorithms; 3) Deciding external priorities of users 4) Collecting information about the average probable needs for resources of a specific job. Pricing, however, should not be a question separated from scheduling. The problems of protection and of deadly embrace had already been separated from the others in previous works. By partitioning the difficulty, I believe that the way to better scientific understanding of shared computer systems stands open.

The previous scheduling algorithms and models apply in computer systems where the shared facilities can either be preempted with very

little overhead (CPU, busses between two levels of fast memory), or cannot be reallocated without a great amount of overhead (memory). They do not apply, however, in cases where a resource can be preempted but the delay imposed on the preempted job is greater than the time during which the preemption occured. This would be the case if, for instance, a job is swapped from the drum into memory, but if at a certain moment it can't get one of the pages because another job has a higher priority to get a page from this sector of the drum, then the preempted job will have to wait an entire revolution of the drum before the opportunity to get the missing page is repeated, and the cost of having a set of pages idling in memory during all that time is of course important. In such a case, the right strategy might be to avoid preemption, and to decide what to do by computing a "desirability ratio" for each possible scheduling operation (ratio of the urgency by the total cost of the resources involved). (see section IV.4).

It is my belief that the scheduling techniques described in this chapter will be especially useful for scheduling of real-time users, who want to have the assurance of getting a certain percentage of usage of the resources of the machine before they start working.

Other investigations of multileveled scheduling are still necessary. I believe that queuing theory gets enormously complicated too rapidly when the number of servers and the complexity of the queueing strategy increase. Simulation is a fast way of testing whether some algorithm is workable, but is not more than a predictive technique. It does not seem to be likely in the future that a scheduler will first simulate the situation before making a decision. Analytical approaches are almost all

that are left to improve schedulers in the future with the certainty that the designed algorithm will work almost optimally in all cases.

CHAPTER IV
SWAPPING ALGORITHMS

In this chapter a study is made of swapping algorithms for a computer with two levels of memory: drum and core. Pre-paging takes place before a program uses the CPU, an entire working set of pages of this program is swapped into core. Section IV.1 presents the Berkeley or Van Tuyl algorithm, which was developed under the direction of Butler Lampson at Berkeley. Then, by contrast, another swapping algorithm is presented in section IV.2. I then explain why I think that the latter algorithm is much more appropriate than the former, especially for future computer systems. Resource utilizations of users programs under both algorithms are compared in section IV.3. Finally, section IV.4 gives some indications as to how a drum to core system should be scheduled if the swapping algorithm of section IV.2 is used.

The various notations used in this chapter are completely independent from those used in the previous chapters.

IV.1 The Swapping Algorithm of Van Tuyl.

[10] describes a swapping algorithm between drum and core which was intended for the BCC-1 computer.[19] The system has essentially four resources:

---

[19] It was initially designed for the SCC 6700 computer.

a) one CPU

b) core memory

c) drum (capacity supposed to be infinite)

d) a channel between core and drum

A program might be in four possible states:

1) on the drum,

2) being brought into core,

3) in core, waiting for or using the CPU, or waiting for an absent page,

4) being swapped out of core, to the drum.

An "external scheduler" decides which programs are candidates to be brought into core, and among those which are in core, which one gets the CPU, or which are candidates to be swapped out.

A program is considered to be in core when a certain set of pages is in core (this set might be the entire program). Programs are supposed to be small enough so that their pages can be retrieved entirely in one drum rotation (if there is not conflict). A conflict occurs if two programs, while both are being brought in, happen to have a page on the same sector of the drum.

The swapper is an algorithm which has to decide, at each sector of the drum, which page should be transferred. It might do either a read, or a write. With Van Tuyl's algorithm (hereafter called "the Berkeley algorithm"), pages which are not dirty (not written on while in core), do not need to be written on the drum. Van Tuyl simulated his algorithm with the assumption that half of the pages of each program are dirtied while in core.

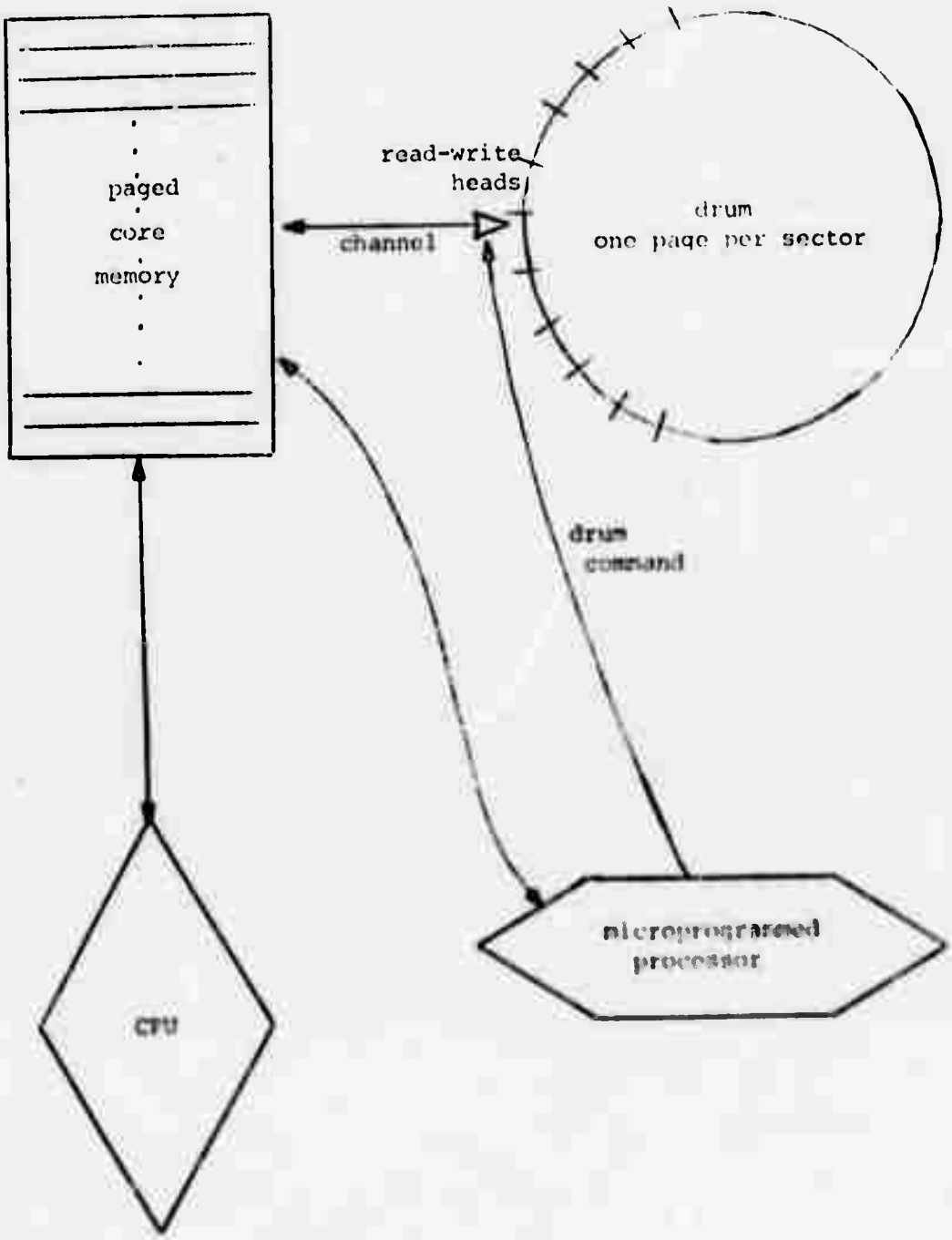


Fig. IV-1

A highly simplified representation of the PDP-1

Table IV-1

Decisions of the Swapper in the Berkeley Algorithm

1. A process that the scheduler has decided to run is put on the swap in list (or read list) if and only if:

   Pages queued in + pages of process to queue in - free core < δ

   (where δ is a system dependent constant), and pages queued out < free slots on read sector list.

2. Drum command:

   a) If no read to do and at least one write to do then write out, exit.

   b) Compute, if conflict (many processes requesting to read a page in on the same sector), for all processes on the read list:

   Cost of process = time to complete reads x (α-PC)

   where PC = number of pages of the process in core

   $\alpha \geq$ # pages a process may have (α is the same for all process, like δ was previously).

   c) If there is a free page in core then read the page of the lowest cost process, exit.

   d) If cost of read > δ or no page can be released in core, then do a write else read the page of the lowest cost process, exit.

3. A page is released if

   page in core belonging to a process p_2 such that

   $\text{cost}(p_2) > \text{cost}(p_1)$

   $\text{cost}(p_2) > \text{cost}(p), \forall p \in \Phi - p_2$

   where p_1 is the "best read" process, and Φ the set of processes candidate to swap in.

A complete description of the swapper operations is given in table IV-1.

Note that the swapper does not "look ahead." When the drum is positioned at a certain sector, the swapper ignores which pages will be candidate to be swapped in on later sectors. This is a reasonable choice because the algorithm is already quite complicated, and one may wonder whether all the decisions of table IV-1 can be made by a micro-programmed processor during the time of one page read (1 millisecond).

IV.2 Another swapping algorithm.

The main idea of this new swapping algorithm is to swap contiguously in time the pages which belong to the Working Set of a program which has to be brought in or out of core. Contrary to the Berkeley algorithm, all of the pages are swapped out (and not only the dirty pages). This somewhat increases the channel utilization, but results in big savings in memory. The reason for these savings is that a small program can now be brought into core in much less than an entire drum revolution since its pages occupy contiguous sectors of the drum. A program is chosen to be swapped in because it has a high external priority and because its set of pages is about to reach the drum heads. To avoid the possibility that such a strategy would indefinitely delay the running of some jobs, the scheduler must decide "in advance" which jobs should be swapped in or out during a drum revolution (see section IV.4).

Note that this new algorithm considerably simplifies the task of the swapper. There is only one job to be swapped in at a time, or one

job to be swapped out. As long as the swapping (in or out) of the current job is not completely finished, there is no possible conflict, and thus the next page to be transferred is always obvious. When the swapping of a job is completed, the next job to be swapped must be selected. It can be any job in core if a swap-out is desired; otherwise, a job must be chosen by the scheduler to be swapped in such that its first drum sector has not yet arrived at the read heads. When initiating a swap-in operation, there must be an assurance that enough free core is available. The problem of determining which job will be swapped in or out, will be studied in section IV.4.

IV.3 Comparison of the resource utilization under both algorithms.

To simplify, suppose that there is just one CPU and one drum. Then, there are three important resources: the CPU, the core memory, and the drum-to-core channel. The drum memory is supposed not to be saturated under normal conditions. The utilization of these three resources is computed by a program, during an entire cycle (swap-in, compute, swap-out). Those utilizations will be normalized in time-utilization of the entire resource. For instance, if the core memory size is M, the use of an amount m of core during time t is normalized to a memory utilization of $\frac{m}{M}$ t.

While computing the resource utilizations, some simplifying assumptions are made. The strongest of them is to neglect the increased utilization of memory and channel due to conflicts in the Berkeley algorithm. Removing this particular assumption strengthens the conclusions which follow even more.

The following definitions will aid in the discussion:

S number of sectors on one drum revolution

P number of pages of a program ($\frac{P}{2}$ of them dirty)

M size of core memory

T time spent by the CPU on a program, while it is in core

S and P both have a time dimension. In these computations, the time unit is the time to read one sector from the drum (1 millisecond on the BCC-1). The resource utilizations are shown in table IV-2.

The channel utilization time is $\frac{3}{2}P$ in the first algorithm, compared to 2P in the second, due to the fact that the dirty pages are not swapped out in the first algorithm.

Memory utilization is computed as a space-time product (see figure IV-2). With the Berkeley algorithm, for instance, the program is brought into core in one drum revolution (time S), handled in time T by the CPU, and swapped out in time $\frac{3}{2}P$ (in the first phase of this swap-out, the clean pages, which do not need to be swapped, are replaced by another program's pages; in the second phase, the $\frac{P}{2}$ dirty pages are swapped, at a rate of one page per two sectors, the other sector time being given to a read). The total resource utilization of a program is now defined as the maximum of the three resource utilizations (channel, core and CPU):

For algorithm #1:

$$U_1 = \max\left(T, \frac{PT + \frac{SP}{2} + \frac{5}{8}P^2}{M}, \frac{3}{2}P\right)$$

For algorithm #2:

$$U_2 = \max\left(T, \frac{PT + 2P^2}{M}, 2P\right)$$

Resource	Resource Utilization	T = 10 P = 10 S = 32 M = 32	T = 10 P = 5 S = 64 M = 10
CPU	T	10	10
memory	$\frac{PT + \frac{SP}{2} + \frac{5}{8}P^2}{M}$	10	23
channel	$\frac{3}{2}P$	15	7.5

Resource	Resource Utilization	T = 10 P = 10 S = 32 M = 32	T = 10 P = 5 S = 64 M = 10
CPU	T	10	10
memory	$\frac{PT + 2P^2}{M}$	9	10.
channel	2P	20	10

Table IV-2

Resource utilization by the swapping algorithms #1 and #2

At the top, the Berkeley algorithm; at the bottom, the other algorithm

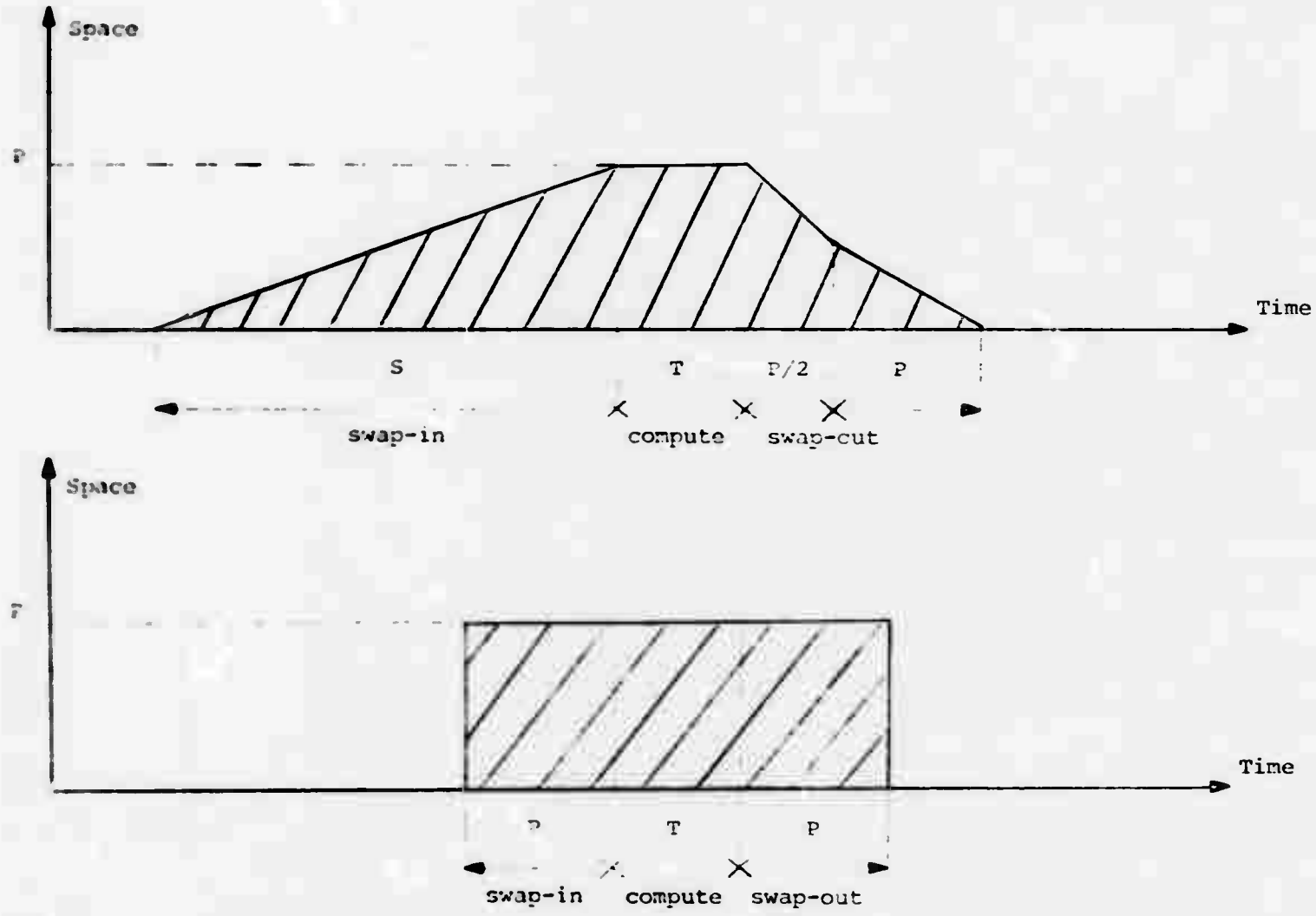


Fig. IV-2 Core memory utilization viewed as an area in the time-space domain
At the top is shown the Berkeley algorithm; below, is the algorithm of section IV.3

The Berkeley algorithm was simulated by Van Tuyl under various assumptions, among them, P = 10 pages, S = 32 milliseconds (or pages), M = 32 pages and T = 10 milliseconds. Table IV-2 shows that $U_1 = 15 < U_2 = 20$ with these data, so that the 1st algorithm really behaves better than the second algorithm, and the bottleneck really lies in the channel. If the memory size and the speed of the channel are decreased to P = 5, S = 64, M = 10, and T = 10, the second algorithm performs much better than the first one: $U_1 = 23 > U_2 = 10$, and the bottleneck of the first algorithm lies in the memory.

Now follows a study of how the resource utilization would change if the characteristics of the available hardware were to change.

The size of the memory, M, the length of a drum revolution, S, or the bandwidth of the channel, B could be varied. B was supposed to be equal to 1 in the previous computations. More generally:

$$U_1 = \max\left(T, \frac{PT + \frac{SP}{2} + \frac{5}{8}\frac{P^2}{B}}{M}, \frac{3}{2}\frac{P}{B}\right)$$

$$U_2 = \max\left(T, \frac{PT + 2\frac{P^2}{B}}{M}, 2\frac{P}{B}\right)$$

1) Effect of bandwidth.

Figure IV-3 shows the effect of bandwidth. For high bandwidth, algorithm #2 performs better than algorithm #1, as expected. Note that this is not true if T were high (in which case both algorithms would be CPU bound); but the assumption is made that CPU's are getting faster and cheaper, and are not the critical resources of modern computer systems.

2) Effect of drum rotation time.

Figure IV-4 shows how, when the drum rotation time increases, algorithm #1 looses efficiency, but algorithm #2 does not degrade at all. This is due to the fact that the memory utilization by algorithm #2 is independent on S. This will allow the possible use of slow, cheap drums in future computer systems.

3) Effect of a change of the relative cost of core memory versus cost of other resources.

In figure IV-5, it can be seen that if the core memory size decreases, the second algorithm does not get memory bound as rapidly as the first one. This will be helpful if memory is the critical resource in the future.

Demand paging. If a page is missing, the normal strategy under algorithm #1, is to leave the program in core while the page is being brought in. But, with algorithm #2, if enough bandwidth is available, it is cheaper to swap the entire working set of the program back onto the drum, to bring the missing page into core, and then to swap the working set in again when it arrives under the read heads of the drum. The same considerations apply for a short I/O operation.

A final word is necessary about the accuracy of these resource usage estimates. It was assumed that there would be no conflict; in other words, for each sector of the drum, there is zero or one page transferred, but never pages of two different jobs both wanting to be transferred. Possible conflicts tend to increase sometimes considerably the resource usage (memory and channel) for algorithm #1, whose actual

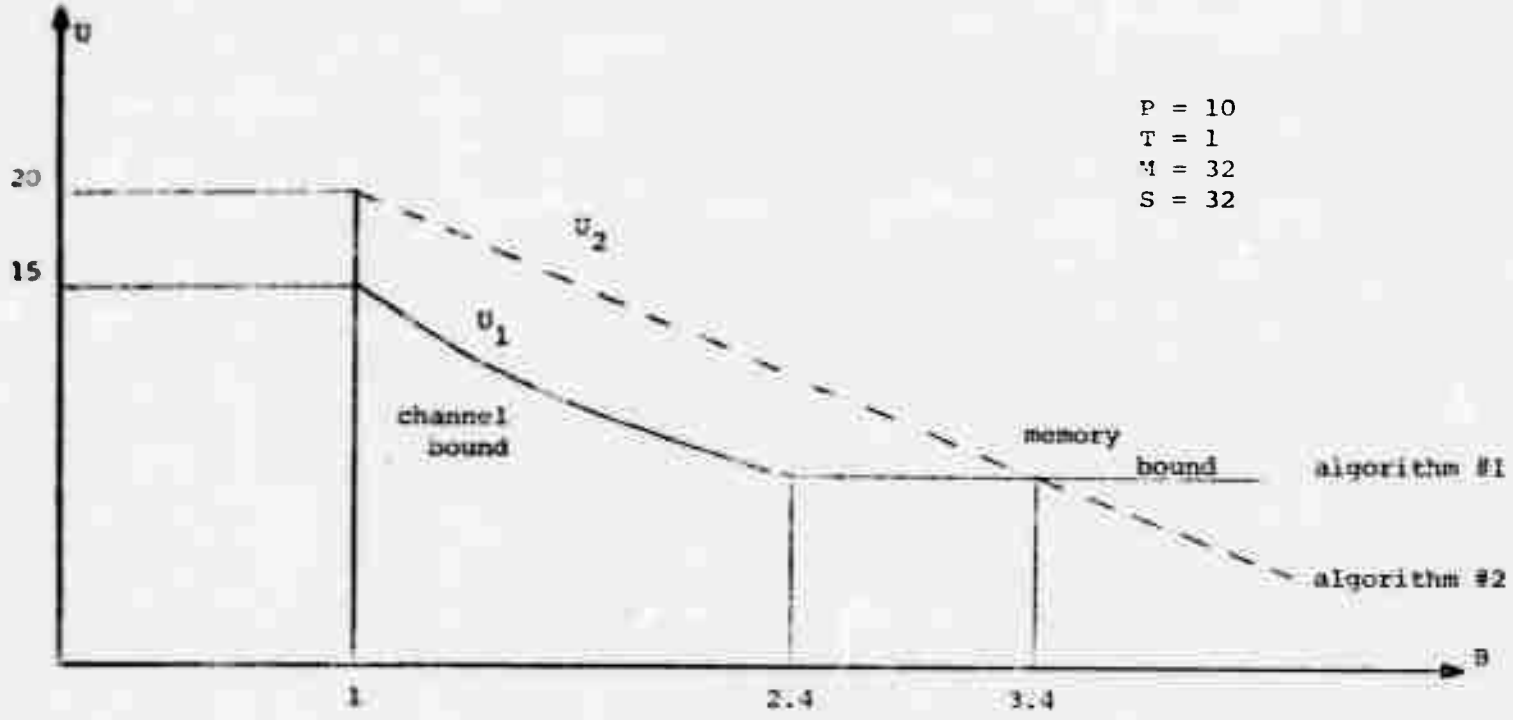


Fig. IV-3

Effect of bandwidth

on total resource utilization of a program

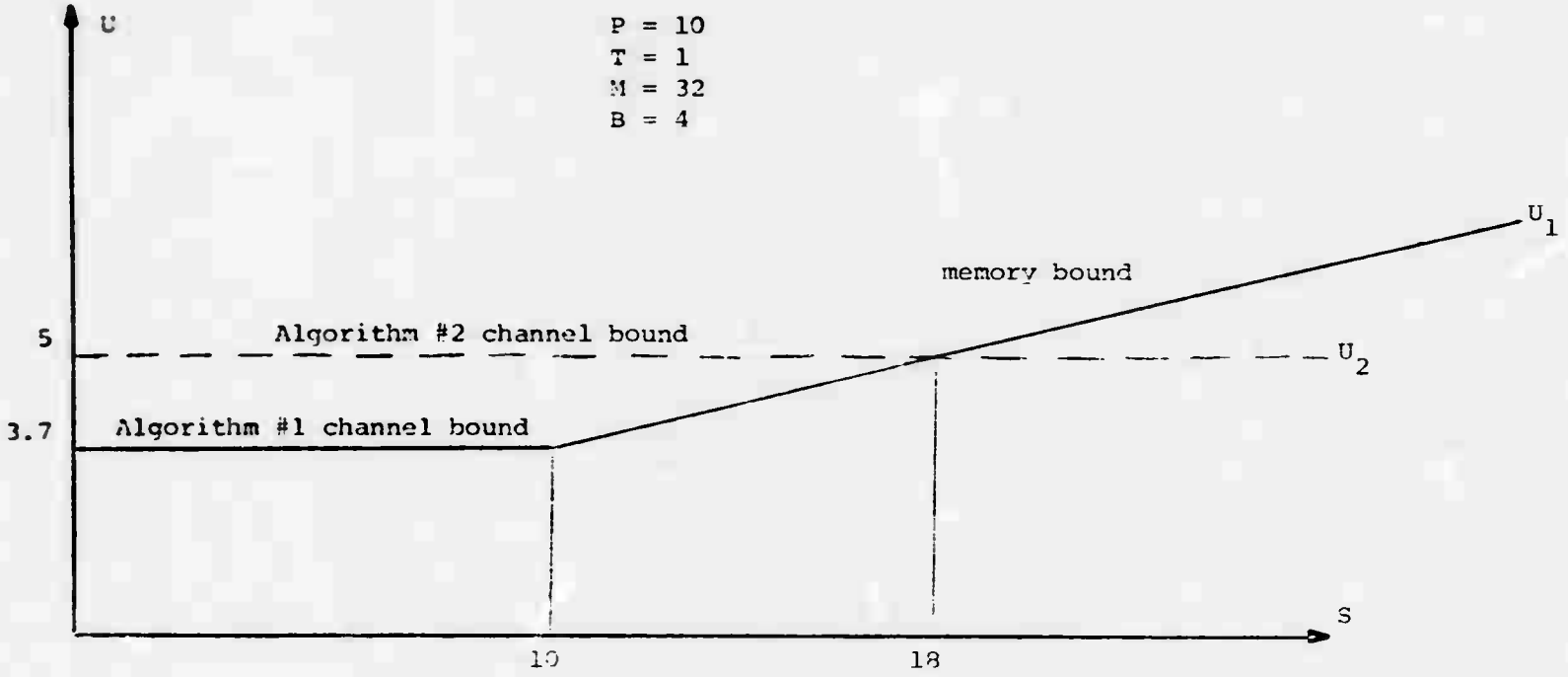


Fig. IV-4

Effect of drum rotation time
on total resource utilization of a program

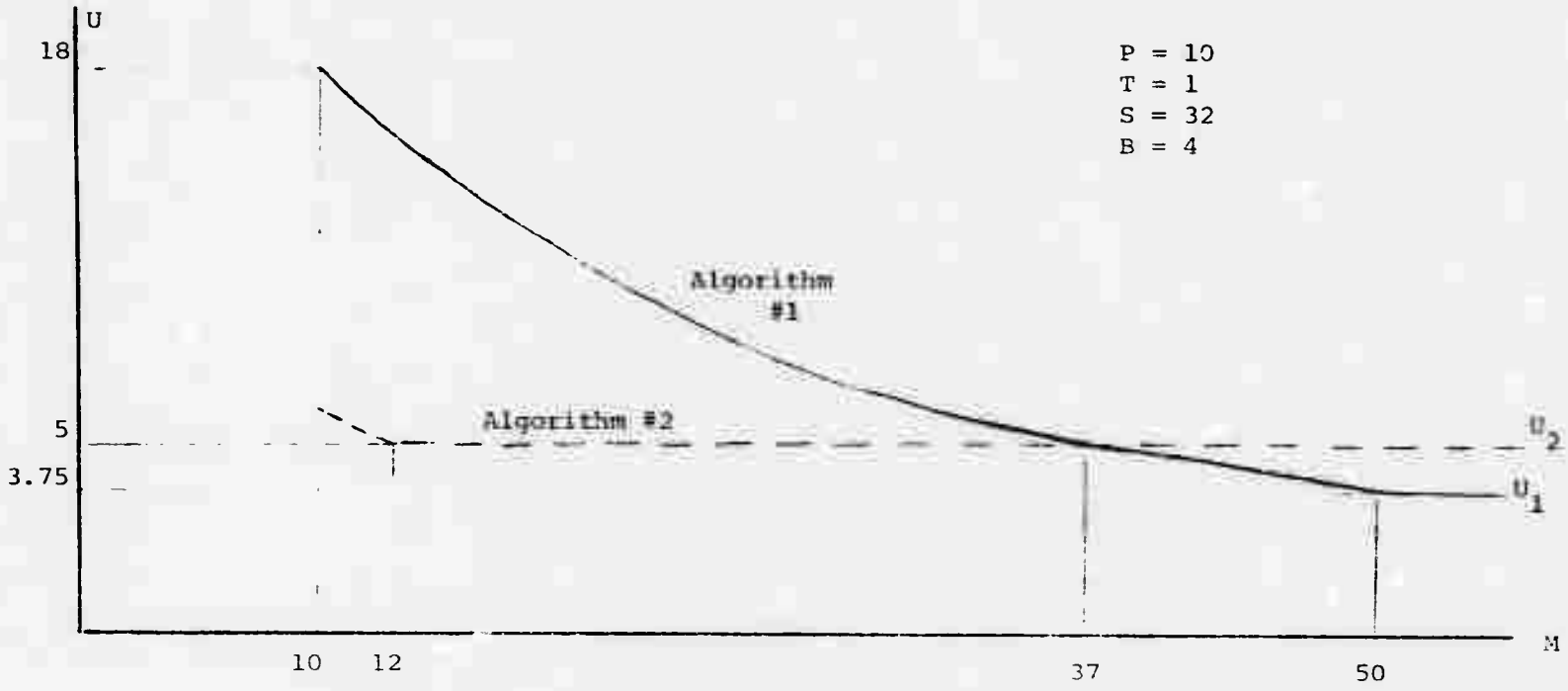


Fig. IV-5

Effect of main memory size

on total resource utilization of a program

behaviour can be much more resource consuming than the figures show. This would lead to a preference for algorithm #2 even more than was previously computed.

IV.4 Scheduling a computer system under the algorithm of section IV.3.

The scheduling problem considered here, consists of deciding which job, at a given time, is to be swapped into core, out of core, or to occupy the CPU.

IV.4.1 Scheduling Criterion.

For user i, we suppose to be known:

1) his bid C_i.

2) his requested CPU utilization time T_i.

C_i is the bid for an entire cycle of swap-in, CPU usage during an interval T_i, and swap-out.

An additional constraint is that a job can only be swapped in when its sectors on the drum pass under the read heads.

The system's criterion for scheduling is to maximize

$$E = \sum_{i \in S} C_i$$

where S is a set of users which can be run over a given time interval.

IV.4.2 Jobs Desirabilities.

It is desirable to allocate jobs in a way such as to get a balance of resource usage (to swap, for instance, a large-sized job while a small-sized job with a large T_i occupies core).

To get such a balanced set of jobs, prices are first assigned to the basic resources:

p_{CPU}

p_{CH}

p_{MEM}

The desirability of job i is then:

$$d_i = \frac{c_i}{a_{CPU} p_{CPU} + a_{CH} p_{CH} + a_{MEM} p_{MEM}}$$

where a_{CPU}, a_{CH}, a_{MEM} are the resource utilizations of job i, which are given in table IV-2, as functions of the characteristics of the job.

IV.4.3 Job scheduling over a time-interval.

No justification is given here to the algorithm which follows. However, the reader will recognize it as a variation of an algorithm of chapter III.

For a given time-interval $[t_1, t_2]$, a schedule is computed by the following procedure. As nearly as possible, jobs are allocated in order of decreasing desirabilities. A job can be scheduled for swapping if its pages on the drum do not share any common sector with any job already scheduled for swapping, and if the jobs for which allocation has already been decided, leave enough memory for the new job to 1) be brought in, 2) wait for the CPU, 3) run, 4) wait for the channel, and 5) swap out.

After the schedule has been worked out for this time interval, the new prices of resources are computed as a function of the old prices and the idle time of the resources. The scheduling algorithm is activated once every (t_2-t_1) units of time. This might be typically a drum rotation time.

Note that the time spent by the algorithm is proportional to n log n + k, where k is a constant and n log n the time necessary to sort the desirabilities of the jobs candidate to run.

CHAPTER V

CONCLUSION AND PROBLEMS FOR FURTHER RESEARCH

This paper has investigated some ways of

1) getting the user to participate in the allocation of computer resources,

2) finding a balanced set of users to more efficiently share these facilities, and

3) improving current swapping algorithms, and finding an algorithm appropriate for a given hardware configuration when the system is designed.

The allocation problem stands wide open for further research. I suggest the following possible directions:

1) Communication between the user and the system. It would be interesting to design a language through which the user could express his knowledge about his program (hoping that it would be useful to the system's resource allocator). This language would express trade-offs between the amount of money that he wants to spend and the response he is going to get, etc... Along these same lines, it would be interesting to find out more about simple ways of analyzing the structure of a program and its characteristics for allocation.

2) Structuring the resource allocation. Users want more and more complicated contracts with the system, which should guarantee them of getting the service that they expect. Wider classes of possible contracts should be investigated.

3) Prices. What is the effect of a price structure on the user's behavior and on the system's efficiency? How could fair prices be determined easily for complicated contracts?

LIST OF REFERENCES

1. Watson, R.W. Introduction to Time-Sharing Concepts. Technical Progress Report no. 28b-6m, Project no. 7010, Shell Development Company (January 1965).

2. Habermann, A.N. Prevention of system deadlocks. Communications of the ACM 12,7 (July 1969), 373-385.

3. Muth, John F. et al. Industrial Scheduling. Prentice Hall, Englewood Cliffs, N.J., 1963.

4. Conway, Maxwell and Miller. Theory of Scheduling. Addison-Wesley, Reading, Massachusetts, 1967.

5. Lassez, Daniel J. Productivity of multiprogrammed computers. Progress in developing an analytic prediction model. Comm. ACM 12, 12 (Dec. 1969), 670-682.

6. Schrage, Linus. The modeling of man-machine interactive systems. Report no. 6745, Department of Economics, University of Chicago, Sept. 1967.

7. McKinney, J.M. A survey of analytical time-sharing models. Computing Surveys 1, 2 (June 1969), 105-116.

8. Wallace, V.L. and Rosen, R.D. Impact of multiprogramming in real-time control systems. Comm. ACM 12,6 (June 1969), 350,358.

9. Belady, L.A. and Kuehner, C.J. Dynamic space-sharing in computer systems. Comm. ACM 12,5 (May 1969), 282-288.

10. Van Dyke, Anthony Hood. An Algorithm for Recovering Data from Drum to Core. Doctoral Diss., ARPA contract #DA-[illegible], Univ. of California at Berkeley.

11. Bolton, Allen. A processor allocation scheme for multi-user on-line operation on a small computer. Proc. AFIPS 1967 Spring Joint Computer Conference, Vol. 30, Thompson Book Co., Washington, D.C., pp. 1-7.

12. Brandon, Peter J. Processor allocation in multiprocessor computer systems. (M.I.T. Thesis), Project MAC, M.I.T., Cambridge, Mass., May 1968.

13. Dennis, Peter J. The working set model for program behavior. Comm. ACM 11, 5 (May 1968), 323-333.

14. Fenichel, Robert R. and Grossman, Adrian J. An analytical model of multiprogrammed computing. Proc. AFIPS 1969 Spring Joint Computer Conference. Vol. 34. AFIPS Press, Montvale, pp. 717-721.

15. Bovet, Daniel Pierre. Memory allocation in computer systems. Report no. 68-17 (Ph.D. Thesis). Computer Science Department. U.C.L.A., June 1968.

16. Nielsen, Norman R. Flexible pricing: an approach to the allocation of computer resources. Proc. AFIPS 1968 Fall Joint Computer Conference, Vol. 33. Thomson Book Co., Washington, D.C., pp. 521-531.

17. Patil, Suhas S. N-server, M-user arbiter. Computation Structures. Group Memo #42, M.I.T. Project MAC.

18. Sutherland, Ivan. A futures market in computer time. Comm. ACM 11, 6 (June 1968) 449-451.

19. Brown, Barbara S. and Gustavson, Frances G. Fortran behaviour in a paging environment. Proc. AFIPS 1968 Spring Joint Computer Conf., Vol. 32. Thomson Book Co., Washington, D.C., pp. 1019-1032.

20. Fuchsman and Randell. Demand paging in perspective. Proc. AFIPS 1968 Spring Joint Computer Conference, Vol. 33. Thomson Book Co., Washington, D.C., pp. 1011-1026.

21. Debreu. Theory of Value. John Wiley & Sons, New York, 1959.

22. Coffman, E.G. and Kleinrock, L. Computer scheduling methods and their countermeasures. Proc. AFIPS 1968 Spring Joint Computer Conference, Vol. 32. Thomson Book Co., Washington, D.C., pp. 11-21.

23. Gue, Ronald L. et al. Analysis of algorithms for the 0/1 knapsack problem. Comm. ACM 11,12 (Dec. 1968), 837-841.

24. Weingartner, Martin H. and Ness, David N. Methods for the solution of the multi-dimensional 0/1 knapsack problem. Oper. Res. 15,1 (January-February 1967), 83-103.

25. Kleinrock, L. Optimum bribing for queue position. Oper. Res. 15,2 (March-April 1967), 304-318.

26. Dijkstra, Edsger W. The structure of "THE" multiprogramming system. Presented at the Symposium on Operating System Principles, Gatlinburg, Tennessee, October 1-4, 1967.

27. Watson, Peter Veitch. The nucleus of a multiprogramming system. Comm. ACM 13,4 (April 1970) 238-241.

28. Everett, Hugh III. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. Oper. Res. 11,3 (May-June 1963), 399-417.

29. Stevens, David P. On overcoming high-priority paralysis in multiprogramming systems, a case history. Comm. ACM 11,8 (August 1968), 539-541.

APPENDIX A

PROOF OF THEOREM 2.

Equations (III-2) can be written, under priority assignment #1:

(A-1) $$\forall k \in [1..n],\ \underline{\text{either}}\ \sum_{k'=1}^{n} \gamma_{kk'} \cdot w_{k'} \leq 1\ \underline{\text{or}}\ w_k = 0$$

$$\text{with } \gamma_{kk'} = \sum_{\rho \in [1..m]} a_{k'\rho} \quad \text{for } k \neq k', \text{ and } \gamma_{kk} = 1$$

$$P_{k'\rho} < P_{k\rho}$$

Under priority assignment #2:

(A-2) $$\forall k \in [1..n],\ \text{either}\ \sum_{k'=1}^{n} \gamma'_{kk'} \cdot w_{k'} \leq 1 \text{ or } w_k = 0$$

$$\text{with } \gamma'_{kk'} = \sum_{\substack{\rho \in [1..m] \\ P'_{k'\rho} < P'_{k\rho}}} a_{k'\rho} \quad \text{for } k \neq k', \text{ and } \gamma'_{kk} = 1$$

It shall be proved that equation #k of (A-1) implies equation #k of (A-2), except for k = 1, and that equation #1' of (A-1) implies equation #1 of (A-2).

1) Note that for $k \neq 1$ and $k \neq 1'$, it is obvious that $\gamma_{kk'} = \gamma'_{kk'}$, because the priorities of these jobs relative to either job 1 or job 1' have not been modified on any processor. This proves equation #k of (A-2) for $k \neq 1$ and $k \neq 1'$.

2) We have: $\gamma_{11} = \gamma_{1'1'} = \gamma'_{11} = \gamma'_{1'1'} = 1$

$\gamma_{11'} = 0$. $\gamma_{1'1} = 1$. $\gamma'_{11'} = a_{ij} \leq 1$. $\gamma'_{1'1} = a_{ij} \leq 1$

thus: $\gamma'_{1'k'} \leq \gamma_{1'k'}$ for any k'; this proves equation #1' of (A-2).

3) Finally, it is shown that equation #1 of (A-2) is implied by equation #1' of (A-1). For $k' \neq 1$ and $k' \neq 1'$, $\gamma'_{1k'} \leq \gamma_{1'k'}$ because $p_{k'j} \leq p_{ij} \rightarrow p'_{k'j} \leq p'_{1'j}$

$$\gamma'_{11'} \leq 1 = \gamma_{1'1'}$$

$$\gamma'_{11} \leq 1 = \gamma_{1'1}$$

QED.

APPENDIX B

PROOF OF THEOREM 3.

The proof is given in three parts:

1. Under a given priority assignment (not necessarily optimal), the maximum of Σ is obtained when the progress rates satisfy:

$$\forall i \text{ . } \underline{\text{either}} \quad w_i = 0 \quad \underline{\text{or}} \quad w_i = 1 - \sum_{\substack{k \\ p_{kj} < p_{ij}}} a_{kj} w_k$$

(This can be proved easily by the reader).

As a consequence, it can be assumed that the jobs are numbered in such a way that: $1 < k \Rightarrow p_{1j} < p_{kj}$ and

$$\begin{cases} \forall i \in [1..n'] & w_i = 1 - \sum_{k<i} a_{kj} w_k \\ \forall i \in [n'+1..n] & w_i = 0 \end{cases}$$

2. If the jobs are numbered as above, then the progress rates are given by:

$$w_1 = 1$$
$$w_2 = 1 - a_{12}$$
$$w_3 = (1 - a_{13})(1 - a_{23})$$
$$\vdots$$

$$w_{n'} = (1 - a_{1j})\,(1 - a_{2j}) \cdots (1 - a_{n-1,j})$$

$$w_{n'+1} = 0$$

$$\vdots$$

$$w_n = 0$$

It will now be shown that if two users i and i+1 having priorities $P_{ij} = P_{i+1,j} - 1$, satisfy

$$\frac{a_{ij}}{c_i} > \frac{a_{i+1,j}}{c_{i+1}}$$

then the priority assignment can certainly be improved by exchanging the priorities of i and i+1 for resource j.

Proof: Under the first priority assignment, the value of the cost function at the maximum is:

$$E_{max} = w_1 c_1 + \cdots + w_i c_i + w_{i+1} c_{i+1} + \cdots + w_n c_n$$

and under the second assignment:

$$E'_{max} = w'_1 c_1 + \cdots + w'_i c_i + w'_{i+1} c_{i+1} + \cdots + w'_n c_n$$

Now obviously:

$$w_k = w'_k \quad \forall k \neq i \text{ and } k \neq i+1$$

so that:

$$E_{max} - E'_{max} = c_i (w_i - w'_i) + c_{i+1} (w_{i+1} - w'_{i+1})$$

$$= (1-a_{1j}) \cdots (1-a_{i-1,j}) [c_i (1 - (1-a_{i+1,j}))$$

$$+ c_{i+1} ((1-a_{ij}) - 1)]$$

so that:

$$E_{max} < E'_{max} \iff c_i a_{i+1,j} > c_{i+1} a_{ij} \iff \frac{a_{ij}}{c_i} < \frac{a_{i+1,j}}{c_{i+1}}$$

3. Now, obviously, any priority assignment different from the optimal priority assignment of theorem 3, is such that there exist two users i and i+1 such that:

$$p_{ij} = p_{i+1,j} - 1$$

and $$\frac{a_{ij}}{c_i} > \frac{a_{i+1,1}}{c_{i+1}}$$

Thus any priority assignment non-trivially different from the one of theorem 3 can be improved.

QED.

APPENDIX C

PROOF OF THEOREM 7.

1) <u>There are only 2 jobs</u>.

Before proving the theorem, the following lemma will be proved:

<u>Lemma</u>: if λ_{12}, λ_{21}, w_1 and w_2 are positive numbers less than 1, then equations

(C-1) $$(1 + \lambda_{12})\, w_1 + \lambda_{21}\, w_2 \leq 1$$

(C-2) $$\lambda_{12}\, w_1 + (1 + \lambda_{21})\, w_2 \leq 1$$

imply

(C-3) $$w_1 + (\lambda_{12} + \lambda_{21})\, w_2 \leq 1$$

Proof: equation (C-3) is achieved by multiplying equation (C-1) by $(1 - \lambda_{12})$, equation (C-2) by λ_{12}, and adding.

Proof of the theorem: Assume that $\forall j, i \neq k \iff a_{ij} \neq a_{kj}$. It must be shown that equations (III-10) imply equations (C-4):

(C-4) $$1 \geqslant w_i + \sum_{a_{kj} < a_{ij}} a_{kj}\, w_k$$

Equations (III-10) may be rewritten as equations (C-1) and (C-2) with:

$$\lambda_{ij} = \sum_{\substack{k \\ a_{ik} w_i < a_{jk} w_j}} a_{ik}$$

According to the lemma, this implies equation (C-3); now note that

$$\sum_{\substack{k \\ a_{ik} < a_{jk}}} a_{ik} \leq \lambda_{ij} + \lambda_{ji}$$

so that equations (C-4) are verified.

2) <u>There is only one resource</u>.

It must be shown that equations (C-5) imply equations (C-6):

$$(\text{C-5}) \qquad \forall i \in [1,n] \ , \ w_i + \sum_{k \neq i} \min (a_i w_i \ , a_k w_k) \lesssim 1$$

$$(\text{C-6}) \qquad \forall i \in [1,n] \ , \ w_i + \sum_{a_k < a_i} a_k w_k \lesssim 1$$

Consider the set S of jobs such that $k \in S \iff a_k w_k > a_i w_i$.

Does there exist a job $k' \in S$ such that $w_{k'} > w_i$?

a) yes, there does. Then choose k' such that there is no job in S whose progress ratio is less than $w_{k'}$ and greater than w_i. Then equation #k' of (C-5) implies equation #i of (C-6).

b) there is no such k'. This means that $\forall k \in S$, $w_k < w_i$, and $a_k > a_i$. Thus equation #i of (C-5) implies equation #i of (C-6).

3) Note that the theorem is not valid for any number of jobs and any number of resources, as shown by the following counter-example:

$$(a_{ij}) = \begin{pmatrix} .25 & .25 & .25 & .25 & & & & & \\ .24 & & & & .76 & & & & \\ & .24 & & & & .76 & & & \\ & & .24 & & & & .76 & & \\ & & & .24 & & & & .76 & \end{pmatrix}$$

The progress rates $w_1 = .5$, $w_2 = w_3 = w_4 = w_5$ .875, satisfy equations(III-10), but not equations(III-2).

However, I suspect that the optimum of $\sum_i w_i$ is always higher with equations(III-2) than with the constraints of equations(III-10).